



Universität Hamburg  
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# **Authentication in Ultra Large Scale REST-based Systems**

Kumulative Dissertation

zur Erlangung des akademischen Grades

Dr. rer. nat.

an der Fakultät für Mathematik, Informatik  
und Naturwissenschaften der Universität Hamburg

eingereicht beim Fach-Promotionsausschuss Informatik von

Hoai Viet Nguyen

Mai 2020

**Gutachter:**

Prof. Dr.-Ing. Hannes Federrath

Prof. Dr.-Ing. Mathias Fischer

Prof. Dr.-Ing. Luigi Lo Iacono

**Tag der Disputation:**

28. Oktober 2020



## Danksagung

Diese Arbeit möchte ich all den Menschen widmen, die mich bei meinem Weg bis zur Verfassung dieser Dissertation begleitet haben. Es war ein langer und steiniger Weg. Ich hätte vor Beginn meiner Hochschullaufbahn niemals daran gedacht, dass ich ein solches Ziel erreichen könnte. Ohne diese Menschen wäre dies nicht möglich gewesen.

Als aller ersten möchte ich mich beim Herrn Prof. Dr. Luigi Lo Iacono bedanken. Ich habe dir so viel so verdanken. Vielen Dank, dass du an mich geglaubt hast. Danke, dass du mir so viele Türen gezeigt und geöffnet hast. Danke für deine Geduld, Ehrlichkeit und fachlichen sowie Lebensratschläge. Als Nächstes möchte ich mich bei Prof. Dr. Hannes Federrath bedanken, der trotz seines vollen Terminplans sich die Zeit genommen hat meine Promotion als Universitätsprofessor zu betreuen. Einen großen Dank gilt auch an die Arbeitsgruppe Security and Privacy der Universität Hamburg, die mich von Anfang an mit offenen Armen empfangen hat. Ich bedanke mich auch bei meinem langjährigen Arbeitskollegen und Freund Peter Leo Gorski, der sich nie davor gescheut hat mir konstruktive und ehrliche Kritik zu geben. Vielen Dank auch an Stephan Wiefeling für das virale Marketing von CPDoS und natürlich für seine konstruktive und ehrliche Anmerkungen meiner Vorträge und Arbeiten. Dieser Dank gilt auch allen weiteren Arbeitskollegen der Gruppe für Daten- und Anwendungssicherheit: Jan Tolsdorf, Florian Dehling, Tobias Mengel, Markus Thomas Marcinek, Paul Höller, Rafeal Mäuer, Daniel Torkian, Peter Nehren und Aline Jaritz.

Zum Schluss möchte ich mich bei meiner Familie bedanken. Ich danke meinen Eltern und meinem Bruder Trung sowie seiner Ehefrau Cherry, die mich immer unterstützt haben. Einen großen Dank gilt auch meinen Schwiegereltern, Co Nhung, Tram, Andre, Clara, Bela und allen anderen Familienmitgliedern. Vor allem möchte ich mich auch bei meiner Frau Yen-Mei und meiner Tochter Emma bedanken, die mir so viel Lebensfreude, Kraft und Energie geben immer wieder aufzustehen und weiterzumachen egal vor welchen Herausforderungen wir stehen.

## Zusammenfassung

Durch die Digitale Transformation sind Softwaresysteme zu einem unverzichtbaren Bestandteil unserer Gesellschaft und Berufsleben geworden. Fast in allen Bereichen des Alltags werden vermehrt verteilte Softwaresysteme eingesetzt, um unser Leben zu erleichtern oder um Geschäftsprozesse zu optimieren. Schon heute umfasst das Internet mehrere Millionen Softwareservices, welche von Milliarden Menschen täglich genutzt werden. Da moderne verteilte Anwendungen eine Vielzahl von Nutzern und Datenträffik verarbeiten, werden sie als massive-skalierbare Softwaresysteme bezeichnet (engl. Ultra Large Scale Systems). Ein bedeutender Ansatz um massive-skalierbare Softwaresysteme zu entwickeln ist der Architekturstil Representational State Transfer (REST). Eines der prominentesten und wichtigsten Insantziierungen dieses Konzept ist das Web, das momentan (wohl) größte massive-skalierbare Softwaresystem der Welt. Durch die hohe Anzahl an Nutzern und die Wichtigkeit von Software für die Gesellschaft und Wirtschaft ist neben der massiven Skalierbarkeit die Sicherheit ein essenzielles Qualitätsmerkmal für REST-basierte Anwendungen.

Um Skalierbarkeit und Sicherheit gewährleisten zu können, verwenden Unternehmen vermehrt Transport Layer Security (TLS) und intermediäre Systeme wie z.B. Caches. Mittlerweile hat sich TLS zu einem festen Bestandteil von Webanwendung etabliert. Auch die Verwendung von intermediären Systemen nimmt immer weiter zu. Content Delivery Networks (CDN), welche ein weltweites Netz an Cachingeinheiten umfassen, können nicht nur die Skalierbarkeit erhöhen, sondern sind auch ein effektiver Schutz gegen Distributed Denial of Service (DDoS) Angriffe. Die Verwendung von TLS und intermediären Systemen ist daher eine Kernkomponente für die Sicherheit und Skalierbarkeit moderne Softwaresysteme. Das Zusammenspiel zwischen TLS und intermediären Systemen hat allerdings einen entschiedenen Nachteil. Intermediäre Systeme müssen die TLS-Verbindung unterbrechen, um Nachrichten interpretieren zu können. Diese Unterbrechung führt dazu, dass kein Ende-zu-Ende Schutz sichergestellt werden kann. Zudem nutzen Unternehmen oft intermediäre Systeme oft von Drittanbietern. Daten, die von einem solchen intermediären System verarbeitet werden, liegen somit außerhalb der Kontrollbereiche der Unternehmen und unter Umständen in einem Land mit unzulässigen rechtlichen Rahmenbedingungen. Moderne Softwaresysteme sind jedoch auf TLS und intermediäre Systeme angewiesen, damit Sicherheit und Skalierbarkeit gewährleistet werden kann. Allerdings muss mit der Verwendung beider Technologien ein Sicherheitsrisiko eingegangen werden.

Diese kumulative Dissertation beschäftigt sich mit diesem Problem. Sie untersucht und erarbeitet REST-basierte Ende-zu-Ende Sicherheitsmechanismen mit einem speziellen Fokus auf Authentifikation und Caching. In den gesammelten Papern werden ausführliche Untersuchungen über REST-basierte Authentifizierungsverfahren und Webcaching dargelegt. Die Ergebnisse zeigen neue Erkenntnisse über die Wechselwirkung zwischen Sicherheit und Skalierbarkeit. Es wird eine neuartige Klasse von Angriffen vorgestellt, welche Cachingssysteme ausnutzt, um die Verfügbarkeit jeglicher Ressourcen einer Webanwendung zu unterbinden. Mehrere Millionen Webseiten waren von diesen Angriffen betroffen. Mithilfe der Erkenntnisse aus den Untersuchungen konnten Gegenmaßnahmen erarbeitet und in Zusammenarbeit mit den betroffenen Organisationen implementiert werden. Basierend auf dem Wissen über Authentifizierung und Webcaching wird ein Authentifizierungsverfahren für REST vorgestellt, welches eine ganzheitlichen Ende-zu-Ende Authentizitäts- sowie Integritätsschutz gewährleistet und zudem mit Webcachingssystemen kompatibel ist. Die empirischen Messungen zeigen, dass ein vollständiger Ende-zu-Ende Schutz sichergestellt werden kann, ohne die Skalierbarkeit zu beeinträchtigen.

## Abstract

With the digital transformation, software systems have become an integral part of our society and economy. In every part of our life, software systems are increasingly utilized to, e.g., simplify housework or to optimize business processes. All these applications are connected to the Internet, which already includes millions of software services consumed by billions of people. Applications which process such a magnitude of users and data traffic requires to be highly scalable and are therefore denoted as Ultra Large Scale (ULS) systems. Roy Fielding has defined one of the first approaches which allows designing modern ULS software systems. In his doctoral thesis, Fielding introduced the architectural style Representational State Transfer (REST) which builds the theoretical foundation of the web. At present, the web is considered as the world's largest ULS system. Due to a large number of users and the significance of software for society and the economy, the security of ULS systems is another crucial quality factor besides high scalability.

To ensure scalability and security, web-based ULS applications mostly use Transport Layer Security (TLS) and intermediate systems such as caches. In recent years, TLS has been established as an indispensable security component of protecting HTTP messages in transit. The usage of intermediate systems has also become an essential ingredient in web applications for providing scalability as well as security. Content Delivery Networks (CDNs), for instance, operate a mesh of interconnected caching edge servers scattered around the world to speed up the page loading time. This distributed network of worldwide caching units is also an effective countermeasure for Distributed Denial of Service (DDoS) attacks, as the multitude of caches can resist against a flood of requests. Because of these reasons, TLS and intermediate systems are vital pillars for any modern REST-based ULS system. The interplay of TLS and intermediate systems, however, has one major drawback. Each intermediary must terminate the TLS connection to read and change traversing messages. Caches, e.g., require to interpret certain message parts to infer the caching policy and the cached content. Such an interruption leads to the issue that messages are not protected from end-to-end. Considering the fact that many organizations use third-party intermediate systems, the full access to traversing messages forces content providers and users to blindly trust intermediaries not to tamper or eavesdrop sensitive content. This is a critical security risk that must be taken by every user and provider of contemporary REST-based ULS applications.

This cumulative dissertation aims at addressing these issues. It studies end-to-end security means with a special focus on authentication and caching. The included papers cover large-scale investigations on REST-based authentication schemes and web caching. The findings show new insight on the interference between security and scalability. The thesis introduces a new class of attack that takes advantage of caches to sabotage all kinds of web resources. Millions of web sites were affected by this threat. Together with the affected parties possible mitigations and countermeasures have been discussed and deployed. With the knowledge from our studies in REST-based authentication and web caching, CREHMA, a cache-aware authentication scheme for high-scalable REST-based web applications is proposed. CREHMA provides end-to-end message authenticity and integrity while being compatible with caches. Experiments show that CREHMA ensures a comprehensive end-to-end security without the loss of scalability.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	List of Publications . . . . .	2
1.1.1	Comments on my Participation . . . . .	3
1.1.2	Other Publications . . . . .	3
1.2	Problem Statement . . . . .	4
1.3	Related Work . . . . .	5
1.4	Research Questions and Contributions . . . . .	6
1.5	Thesis Structure and Research Methodologies . . . . .	8
<b>2</b>	<b>On the Need for a General REST-Security Framework</b>	<b>12</b>
2.1	Introduction . . . . .	13
2.2	REST Foundations . . . . .	14
2.3	Methodology . . . . .	17
2.4	REST-Security Demands and Specifics . . . . .	18
2.4.1	SOAP-based Web Services Security Stack . . . . .	18
2.4.2	REST-ful Services Security Stack . . . . .	19
2.4.3	REST-Security Specifics . . . . .	20
2.5	Related Work Analysis . . . . .	22
2.5.1	HTTP Basic and HTTP Digest Authentication . . . . .	22
2.5.2	API-key . . . . .	23
2.5.3	HOBA . . . . .	23
2.5.4	HTTP SCRAM . . . . .	24
2.5.5	Mutual Authentication Protocol for HTTP . . . . .	24
2.5.6	De Backere et al. . . . .	25
2.5.7	Peng et al. . . . .	25
2.5.8	FOAF+SSL/WebID . . . . .	25
2.5.9	Google, Hewlett Packard and Microsoft . . . . .	26
2.5.10	Amazon . . . . .	26
2.5.11	Signing HTTP Messages . . . . .	27
2.5.12	OAuth . . . . .	27
2.5.13	Serme et al. . . . .	28
2.5.14	Lee et al. . . . .	28
2.5.15	OSCORE . . . . .	29
2.5.16	Granjal et al. . . . .	30
2.5.17	Consolidated Review of Analysis Results . . . . .	30
2.6	Towards a General REST-Security Framework . . . . .	32
2.6.1	Formal Description of REST Messages . . . . .	33
2.6.2	REST Message Authentication (REMA) . . . . .	34

2.6.3	REST Message Confidentiality (REMC)	37
2.7	Implementation of REST Message Authentication	39
2.7.1	REST-ful HTTP Message Authentication (REHMA)	39
2.7.2	REST-ful CoAP Message Authentication (RECMA)	40
2.8	Evaluation and Discussion	42
2.9	Conclusion and Outlook	44
<b>3</b>	<b>On the Security Expressiveness of REST-based API Definition Languages</b>	<b>46</b>
3.1	Introduction	47
3.2	Representational State Transfer (REST)	48
3.3	Security Schemes for REST-based Web Services	48
3.4	Description Languages for REST-based Web Services	50
3.4.1	WSDL	50
3.4.2	WADL	51
3.4.3	RSDL	51
3.4.4	RADL	51
3.4.5	REST Chart	51
3.4.6	OAS / Swagger	52
3.4.7	RAML	52
3.4.8	API Blueprint	53
3.4.9	OData	53
3.4.10	I/O Docs	53
3.4.11	hRESTS and RDFa	54
3.4.12	ReLL	54
3.4.13	SERIN	55
3.4.14	Hydra	55
3.4.15	RESTdesc	55
3.5	Discussion	56
3.6	Conclusion and Outlook	57
<b>4</b>	<b>Systematic Analysis of Web Browser Caches</b>	<b>59</b>
4.1	Introduction	60
4.2	Web Caching Foundations	60
4.2.1	Explicit Caching	61
4.2.2	Implicit Caching	64
4.2.3	Client-originated Caching Policies	64
4.2.4	Defining new Cache Keys	64
4.2.5	Invalidation of Freshness	65
4.2.6	Partial Content	65
4.2.7	Security	65
4.3	Web Browser Caches	65
4.4	Related Work	66
4.5	Test Methodology	66
4.6	Implementation of Testing Tool	68
4.6.1	Web Browser Cache Testing Tool	68
4.6.2	Test Case Syntax	69
4.7	Empirical Study	71
4.8	Conclusion	73



<b>5</b>	<b>Mind the Cache: Large-Scale Explorative Study of Web Caching</b>	<b>74</b>
5.1	Introduction	75
5.2	Web Caching	76
5.2.1	Freshness	77
5.2.2	Client-originated Policies	78
5.2.3	Cache Key Adaption	79
5.2.4	Invalidation of Freshness	79
5.2.5	Partial Content	79
5.2.6	Security	80
5.3	Consequences of Malfunctioning Caching	80
5.4	Related Work	81
5.5	Cache Testing Tool	82
5.5.1	Architecture	83
5.5.2	Test Case Suite	83
5.6	Empirical Study Results	86
5.6.1	Freshness Lifetime	87
5.6.2	Freshness Validation	88
5.6.3	Client-originated Caching Policies	89
5.6.4	Invalidation of Freshness	89
5.6.5	Security	90
5.7	Conclusion and Outlook	92
<b>6</b>	<b>Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack</b>	<b>93</b>
6.1	Introduction	94
6.2	Foundations	95
6.3	Security Threats in Web Caching Systems	96
6.4	Poisoning Web Caches with Error Pages	98
6.4.1	HTTP Method Override (HMO) Attack	99
6.4.2	HTTP Header Oversize (HHO) Attack	101
6.4.3	HTTP Meta Character (HMC) Attack	102
6.5	Practicability of CPDoS Attacks	103
6.5.1	Experiments Setup	103
6.5.2	Feasibility of HMO attacks	104
6.5.3	Feasibility of HHO attacks	105
6.5.4	Feasibility of HMC attacks	107
6.5.5	Consolidated Review of Analysis Results	107
6.5.6	Practical Impact	109
6.5.7	Practical Considerations	111
6.6	Responsible disclosure	112
6.7	Discussion	113
6.8	Countermeasures	114
6.9	Conclusion and Outlook	116
<b>7</b>	<b>CREHMA: Cache-aware REST-ful HTTP Message Authentication</b>	<b>120</b>
7.1	Introduction	121
7.2	Web Caching Background	122
7.3	Evolved Threat Model	124
7.4	Review of HTTP Signature Schemes	125

7.5	Requirements for Cache-aware HTTP Signature Schemes . . . . .	128
7.6	CREHMA . . . . .	128
7.6.1	Signature Generation . . . . .	129
7.6.2	Signature Verification . . . . .	130
7.6.3	Use Cases . . . . .	131
7.6.4	Limitations . . . . .	135
7.7	Evaluation . . . . .	136
7.7.1	Compatibility . . . . .	136
7.7.2	Performance . . . . .	137
7.7.3	Security . . . . .	140
7.8	Conclusion and Outlook . . . . .	141
<b>8</b>	<b>Summary and Further Work</b>	<b>142</b>
	<b>Bibliography</b>	<b>145</b>
	<b>Appendices</b>	<b>159</b>
<b>A</b>	<b>Cache Testing Framework</b>	<b>160</b>
<b>B</b>	<b>CREHMA</b>	<b>161</b>

# Chapter 1

## Introduction

Digital products and services have evolved as an integral part of society and the economy. Modern technologies change the way we live and creates new approaches to optimize business processes. This revolution –denoted as the digital transformation [GR15]– leads to a continuous increase in the number of Internet-connected software-intensive systems as well as users of such applications [Sch17]. By now, the Internet already includes millions of software services that are used by billions of people [Boo18]. Emerging trends, including the Internet of Things (IoT) [Car+18] and Cyber-Physical Systems (CPS) [Raj+10] additionally enrich the sheer magnitude of software services and human users with millions of interconnected devices. In this respect, the term Ultra Large Scale (ULS) [Fei+06] systems has evolved as a new generation of modern distributed software systems covering unprecedented volumes of transferred data, numbers of different software and hardware as well as human participants.

Roy Fielding has introduced one of the first approaches for implementing ULS systems [Xu+08]. In his doctoral thesis, Fielding defines the architectural style Representational State Transfer (REST) [Fie00]. To ensure scalability, REST proposes a set of architectural constraints. According to REST, high-scalable software systems must be layered. This means, besides client and server, software architectures must be composed of intermediate systems (also known as middleboxes [CB02]), such as caches, load balancers, message routers or Web Application Firewalls (WAF). The usage of intermediaries ensures scalability, as data traffic and workload load can be outsourced to other components. Due to the utilization of intermediate systems, communication in REST-based systems must be stateless and messages need to be cacheable. Stateless communication means that request messages must be self-descriptive in the sense that they contain all of the necessary information so that middleboxes and endpoints understand the content without the need to store any state. As state information does not have to be stored, servers can quickly free resources after processing a request and use the available capacity to process other messages. Caching ensures scalability as well. Cacheable messages can be stored and reused by intermediate systems without the need to communicate with the origin server. This eliminates interactions and reduces network traffic as well as the end user-perceived latency.

The most prominent instantiation of REST is the web, the world’s largest distributed system to date. The core technology of the web is Hypertext Transfer Protocol (HTTP) [FR14b], a REST-ful application layer protocol. Due to the benefits of REST and its architectural principles, it is applied in many other areas of distributed computing. This includes, e.g., IoT or CPS. As devices in IoT and CPS are constrained in computation power and network bandwidth, the Constraint Application Protocol (CoAP) [SHB14] has been standardized as an HTTP-based

transfer protocol for high-scalable resource-restricted environments. Other application domains in which REST gets adopted are Service-Oriented Architectures (SOA) [Erl+13], Microservices [New15], Cloud Computing [EPM13a], Smart-\* [PTG14] and Industry 4.0 [WSJ17] applications. Moreover, the fifth generation of mobile communication systems (5G) utilizes REST-based web technologies for implementing the Service-Based Architecture (SBA) covering core network functions [Mao+]. As these application domains of REST represent vital pillars of the modern economy and society, all of them are considered as mission-critical ULS systems. The security of such systems is, therefore, of paramount importance.

This thesis aims at studying the security of REST-based ULS systems with a special focus on authentication and caching. In particular, it explores available REST-based authentication schemes and the compliance of web caching systems. Based on the discovered compliance issues, this work introduces and evaluates a novel attack vector that exploits web caches to conduct denial of service attacks. With the knowledge of authentication and web caching, an authentication scheme is proposed that ensures end-to-end authenticity and integrity while enabling content providers to enjoy the benefits of caching.

## 1.1 List of Publications

During the research on the security of REST-based ULS systems, fourteen peer-reviewed research papers have been published. Six of these publications listed below are part of this thesis.

1. L. Lo Iacono, H. V. Nguyen, and P. L. Gorski. *On the Need for a General REST-Security Framework*. In: *Future Internet* 11.3 (2019). URL: <https://doi.org/10.3390/fi11030056>
2. H. V. Nguyen, J. Tolsdorf, and L. Lo Iacono. *On the Security Expressiveness of REST-Based API Definition Languages*. In: *International Conference on Trust and Privacy in Digital Business (TrustBus)*. 2017. URL: [https://doi.org/10.1007/978-3-319-64483-7\\_14](https://doi.org/10.1007/978-3-319-64483-7_14)
3. H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Systematic Analysis of Web Browser Caches*. In: *2nd International conference on Web Studies (WS)*. 2018. URL: <https://doi.org/10.1145/3240431.3240443>
4. H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Mind the Cache: Large-Scale Analysis of Web Caching*. In: *34rd ACM/SIGAPP Symposium on Applied Computing (SAC)*. 2019. URL: <https://doi.org/10.1145/3297280.3297526>
5. H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack*. In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019. URL: <https://doi.org/10.1145/3319535.3354215>
6. H. V. Nguyen and L. Lo Iacono. *CREHMA: Cache-ware REST-ful Authentication Scheme*. In: *10th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2020. URL: <https://doi.org/10.1145/3374664.3375750>

### 1.1.1 Comments on my Participation

**Paper 1** This publication is co-authored by Luigi Lo Iacono and Peter Leo Gorski. I was responsible for writing, structuring, visualization, designing the methodology, formal as well as practical analysis and discussing the study results. Luigi Lo Iacono and Peter Leo Gorski took part in writing, structuring, visualization, reviewing and editing the paper.

**Paper 2** In this paper, I was mainly responsible for writing, structuring, visualization, defining the methodology and criteria for assessing the related work. Jan Tolsdorf conducted the related work analysis and wrote a draft version of the paper. Luigi Lo Iacono revised and refined the paper.

**Paper 3 and Paper 4** I was the main author of both papers. I developed the methodology based on an in-depth literature review and I was responsible for visualization and structuring the paper. Moreover, I conducted the practical evaluation and discussed the results. Luigi Lo Iacono and Hannes Federrath provided feedback and refined the papers.

**Paper 5** In this paper, I was mainly responsible for discovering the presented attack. I designed and implemented the methodology to analyze the practicability of the introduced vulnerability. To mitigate this threat, I worked together with affected parties and recommended countermeasures based on the cooperation. Luigi Lo Iacono helped me in the responsible disclosure process. He and Hannes Federrath also revised the paper and provided feedback.

**Paper 6** I was the main author of this work and conducted the related work analysis. Moreover, I defined the methodology and performed all empirical evaluations in this paper. Luigi Lo Iacono revised and edited the paper and provided feedback.

### 1.1.2 Other Publications

The following four papers are not included in thesis, since their contributions are summarized and extended by Paper 1 (Journal paper).

- P. L. Gorski, L. Lo Iacono, H. V. Nguyen, and D. B. Torkian. *Service Security Revisited*. In: *11th IEEE International Conference on Services Computing (SCC)*. 2014, pp. 464–471. URL: <https://doi.org/10.1109/SCC.2014.68>
- L. Lo Iacono and H. V. Nguyen. *Authentication Scheme for REST*. in: *International Conference on Future Network Systems and Security (FNSS)*. Springer International Publishing, 2015. URL: [https://doi.org/10.1007/978-3-319-19210-9\\_8](https://doi.org/10.1007/978-3-319-19210-9_8)
- H. V. Nguyen and L. Lo Iacono. *REST-ful CoAP Message Authentication*. In: *International Workshop on Secure Internet of Things (SIoT), in conjunction with the European Symposium on Research in Computer Security (ESORICS)*. 2015. URL: <https://dx.doi.org/10.1109/SIoT.2015.8>
- H. V. Nguyen and L. Lo Iacono. *RESTful IoT Authentication Protocols*. In: *Mobile Security and Privacy - Advances, Challenges and Future Research Directions*. 1st ed. Advanced Topics in Information Security. Elsevier/Syngress, 2016, pp. 217–234. URL: <https://doi.org/10.1016/B978-0-12-804629-6.00010-9>

Furthermore, I contributed to the following four research papers that are not included in this thesis:

- L. Lo Iacono, H. V. Nguyen, T. Hirsch, M. Baiers, and S. Möller. *UI-Dressing to Detect Phishing*. In: *IEEE 6th International Symposium on Cyberspace Safety and Security (CSS)*. 2014. URL: <https://dx.doi.org/10.1109/HPCC.2014.126>
- P. L. Gorski, L. Lo Iacono, H. V. Nguyen, and D. B. Torkian. *SOA-Readiness of REST*. in: *3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer International Publishing, 2014. URL: [https://doi.org/10.1007/978-3-662-44879-3\\_6](https://doi.org/10.1007/978-3-662-44879-3_6)
- L. Lo Iacono and H. V. Nguyen. *Towards Conformance Testing of REST-based Web Services*. In: *11th International Conference on Web Information Systems and Technologies (WEBIST)*. 2015. URL: <https://doi.org/10.5220/0005412202170227>
- H. C. Rudolph, A. Kunz, L. Lo Iacono, and H. V. Nguyen. *Security Challenges of the 3GPP 5G Service Based Architecture*. In: *IEEE Communications Standards Magazine* 3.1 (2019), pp. 60–65. URL: <https://doi.org/10.1109/MCOMSTD.2019.1800034>

They all deal with topics related to REST and the security of REST-based systems.

## 1.2 Problem Statement

With the rise of digital technologies in every part of our life, software systems are increasingly becoming the target of financially and politically motivated attacks. A recent security report of Akamai observed eight billion attacks on web applications during November 2017 and December 2019 [Her+]. Such an observation should alarm every organization to consider security as an indispensable part of every software development and deployment process. Projects such as Let’s Encrypt and recent actions of major web browser vendors have already made efforts towards this direction [Aas+19; Sch18]. Thanks to both initiatives Transport Layer Security (TLS) [Res18] has been established as an obligatory protection means for ensuring confidentiality, integrity, and authenticity of HTTP messages in transit. In addition to TLS, many content providers rely on intermediate systems as complementary safeguards for threats that are outside the scope of transport layer protection means. Besides providing scalability, caches ensure an increased availability as they can satisfy requests when the origin server is down for some reason. Content Delivery Networks (CDNs), which operate a worldwide mesh of interconnected caching edge servers, cannot only speed up the page loading time but are also an effective countermeasure against Distributed Denial of Service (DDoS) attacks [Gil+16]. Many CDNs also include WAFs for filtering malicious requests such as Cross-Site Scripting (XSS) or SQL injection attacks.

Recent studies observed tremendous growth in the usage of protection means. In recent years, content providers are increasingly using TLS and caches to optimize the security and scalability of web applications. According to Mozilla, the average volume of TLS-encrypted traffic in the web has surpassed the amount of unencrypted data [Cal+19; Fel+17]. Guo et al. observe that 74% of the Alexa top 1K websites utilize CDNs [Guo+18]. Both observations highlight that TLS and intermediate systems have been recognized as a cornerstone of security and scalability, respectively. However, the utilization of transport security means, in conjunction with intermediate systems has one major drawback. To fulfill their duties such as caching messages or filter malicious content, intermediate systems require read access and write access to traversing messages. To do so, the secure connection provided by TLS must be terminated and renegotiated by each intermediary. This means the security of traversing messages does not reach from end-to-end and intermediate systems have full access to the content. Content

providers, therefore, must blindly trust middleboxes to not to tamper or eavesdrop the traversing content. In addition to this, content providers and users have to consider that messages often traverse multiple intermediate systems that are operated by third-party services. Both parties must not expect that all of them can be fully trusted. Moreover, the deployment of TLS as the only pillar is by far not sufficient [Cal+19] as many attacks on TLS have been revealed recently [HSS15].

Man-in-the-middle attacks such as tampering or eavesdropping content are, however, not the only serious threat in layered systems. Web cache poisoning attacks, for instance, are becoming a severe threat for REST-based ULS systems likewise. This kind of vulnerability belongs to the semantic gap attacks which exploit the disparity of two or more parties in interpreting the same object. Such a gap mainly occur in layered systems. Here, the likelihood that components parse the same message differently is very high, as endpoints and intermediate systems are implemented for other purposes but also with different programming environments. In a web cache poisoning attack, a malicious client sends a malformed request which confuses the cache and server so that a harmful response is injected into the cache. Benign clients retrieving the target resource receive the malicious response instead of the genuine one. Chen et al. and James Kettle demonstrated that millions of websites were be affected by web cache poisoning attacks [Che+16; Ket19a; Ket18c]. The researchers showed that even a flawless TLS connection and fully trusted intermediate systems with a WAF as well as DDoS protection cannot avoid this threat. To mitigate man-in-the-middle as well as web cache poisoning attacks, software systems require additional security approaches to TLS, WAF, and DDoS protection.

### 1.3 Related Work

The issue that TLS is mandatory but not sufficient is a known research problem [Cal+19]. Paper 1 shows that many attempts have been published to address this shortcoming. HTTP signatures schemes have been the first approach which aims to provide end-to-end security in layered systems. Here, the whole message is protected by concatenating the header and a body to a string, which is digitally signed. The signature value in conjunction with meta information is then included in a new header entry. Based on this header, the verifying endpoint can validate the authenticity and integrity of the message. Amazon, Google, Hewlett-Packard (HP), and Microsoft, for instance, use HTTP signature schemes as an authentication scheme for API requests [Ama19b; Goo17; Mic17; Hew14]. HTTP signatures schemes have not been widely used as other protection means such as TLS, WAFs, or DDoS defenses. However, there is a growing interest in adopting HTTP signature schemes as an IETF standard [Run19; CS19; BRS19]. Similar efforts in promoting HTTP signature schemes have also been made by academia [Ser+12; LJK15]. This also true for CoAP. As with HTTP, a set of CoAP signature schemes have been presented by standardization bodies as well as academia [GMS13; Sel+18].

The bunch of available work in signature schemes for HTTP and CoAP emphasizes the huge demand for end-to-end protection means. However, in state-of-the-art analysis in Paper 1 shows that signature schemes are very diverse even though they follow the same objective. Moreover, there are only signature schemes available for HTTP and CoAP. In recent years, REST has been established as a universal approach to design ULS systems for various application domains. Therefore, it is plausible to believe that more REST-based protocols will be specified in the future. The Remote APDU Call Secure (RACS) [Uri19], for instance, is a REST-based protocol for microcontrollers, which is currently in the standardization process. No end-to-end security



schemes for RACS are available so far. The state-of-the-art analysis in Paper 1 also reveals that current HTTP and CoAP signature schemes still contain too many vulnerabilities. Amazon, Google, HP, and Microsoft, for instance, only protect the request message. The response message is unprotected. Also, many HTTP and CoAP signature schemes do not include all security-critical headers in the signature process, which opens the door for, e.g., man-in-the-middle attacks on the response body and caching policy. Another issue related to caching is the fact that available end-to-end security schemes barely support web caching systems. As with an attacker, caches change and replay messages. Unlike the malicious man-in-the-middle, caches intervene in the message exchange to optimize the scalability. Such a legitimate intervention is not in conformance with many HTTP and CoAP signatures schemes as they are designed to classify any change or reuse of a signed message as a malicious modification or replay attack. As a consequence, the caching of frequently requested content to optimize scalability must be disabled when end-to-end security is required. On the contrary, REST-based ULS systems which require caches to deliver high-quality content to millions of users are forced to omit end-to-end security schemes.

This thesis aims to address these issues. The goal of this work is to enhance security while retaining the cacheability in REST-based ULS systems. To do so, the limitations of available end-to-end authentication schemes as well as the interference between caching and security approaches need to be well-studied in order to understand the challenges in developing robust and stable protection means for REST-based ULS systems.

## 1.4 Research Questions and Contributions

This dissertation focuses on answering the following research question to address the discussed issues and challenges.

**RQ1:** *How can we define REST security components that are generally valid for any REST-compliant technologies?*

**RQ2:** *How can we design a methodology to derive a comprehensive end-to-end authentication scheme that is in alignment with the principles of REST?*

**RQ3:** *To what extent do caches affect the scalability and authentication of REST-based systems?*

To answer the three research questions, several research methods and empirical studies have been conducted and analyzed. The major results and findings are grouped into four major contributions, which are briefly recapped below.

### **Analysis of available work in security for REST-based systems (Paper 1 and 2)**

As initial work to study the security of REST-based ULS systems, this thesis conducts a large-scale analysis of REST-based authentication schemes (see Paper 1). Based on a threat model and a policy on assets to be protected, 21 security mechanisms have been evaluated. The study points out that existing REST-based authentication schemes contain many vulnerabilities. Many end-to-end authentication schemes protect the request only. Other protection means which consider requests and responses do not protect security-critical message headers. Such an omission opens the door for man-in-the-middle attacks even though the message is protected.



In the second state-of-the art analysis in Paper 2, this thesis provides an evaluation of the expressiveness of REST-based service description languages in terms of specifying security mechanisms. Service description languages are essential auxiliaries for robust and stable machine-to-machine communication. ULS application domains such as IoT, CPS, SOA, Cloud Computing, and Smart-\* are greatly relying on this paradigm. The investigation found out that available service description languages are limited in terms of describing REST-based security schemes. The languages are constrained in describing a fixed set of authentication and authorization schemes. Moreover, almost all do not provide any native options to add further protection means. Only one approach allows extending the description language by other security schemes.

### **General REST-Security Framework (Paper 1)**

Based on the vulnerabilities and limitations found in the related work analysis on authentication schemes, this dissertation discusses the need for a more methodical and REST-compliant approach to security. Therefore, a methodology is proposed that aims to design REST-Security mechanisms based on the same abstraction level as REST itself (see Paper 1). In particular, the framework is designed as a general guideline that is not bound to any specific technology. This security guideline then can be adapted to any suitable REST-ful technologies, e.g., HTTP and CoAP, so that any implementation is based on the same security baseline. Following this approach, this thesis proposes REST Message Authentication (REMA), a generic security scheme that provides end-to-end message integrity and authenticity for REST-based systems. REMA then serves a guideline for deriving concrete signature schemes for REST-ful application layer protocols, including HTTP and CoAP. As a result, Paper 1 introduces REST-ful HTTP Message Authentication (REHMA) and REST-ful CoAP Message Authentication (RECMA).

REHMA and RECMA ensure end-to-end authenticity and integrity but do not provide any confidentiality protection for REST messages. End-to-end message confidentiality is of paramount importance for ULS systems as it ensures that sensitive information is not disclosed to malicious or unauthorized intermediate systems. However, simply encrypting the whole message with the aim that only the client and server can read the content does not comply with the REST principles. REST requires messages to be self-descriptive across all intermediate layers and endpoints to enable stateless communication. The plain encryption of the whole message destroys the self-descriptiveness. To develop end-to-end confidentiality schemes which are in alignment with REST principles, the thesis discusses challenges and initial steps towards the development of a REST message confidentiality scheme based on same methodology for deriving REHMA and RECMA.

### **Large-scale analysis of web caching and Cache-Poisoned Denial of Service (Paper 3, 4 and 5)**

As mentioned above, REST constraints messages need to be self-descriptive in order to enable intermediate processing and stateless communication [Fie00]. Intermediate systems such as caches strongly depend on the self-descriptiveness of messages as they need to interpret certain message parts in order to identify reusable content. To develop security schemes which are in conformance with the REST principles, the knowledge on intermediate systems and caches in

particular need to be well-understood in order to avoid conflicts between security and scalability-enabling technologies.

To study the impact of intermediate systems on the security and scalability of REST-based systems, this thesis provides a large-scale study on web caching. Paper 3 conducts an in-depth literature review on web caching. Based on this knowledge, we proposed a test framework including 415 test cases and a cache testing tool. The whole test suite with all test cases as well as the cache testing tool can be freely downloaded as open-source software. More details about the test framework are included in the Appendix A. With the test suite and testing tool, Paper 3 analyzes the four web browser caches of Chrome, Safari, Firefox and Microsoft Edge. Paper 4 evaluates 6 proxy caches and a CDN. Both papers found many inconsistencies among the caches as well as many non-conformances, which may lead to potential vulnerabilities.

Further investigations have been conducted to study whether the detected malfunctions can be exploited to perform practical attacks on real-world ULS systems. Paper 5 presents a novel class of web cache poisoning attacks. The discovered attack has been coined as Cache-Poisoned Denial of Service (CPDoS). With CPDoS, malicious clients can sabotage the access to all kinds of resources. An empirical analysis of real-world caches shows that millions of web sites were affected. All findings have been reported to the respective parties. Moreover, Paper 5 discusses countermeasures which have been elaborated in cooperation with affected vendors and other researchers.

### **Designing a Cache-ware REST-ful HTTP Message Authentication Scheme (Paper 6)**

The knowledge gained from the studies on web caching led to the conclusion that available HTTP and CoAP signatures do not comply with the caching principles of REST. This also includes REHMA and RECMA. As a result, Paper 6 extends REHMA to make it cache-ware (see Paper 6). This extension has been coined as Cache-ware REST-ful HTTP Message Authentication (CREHMA). With the prototype implementation of CREHMA, Paper 6 evaluates its compatibility with existing web caches, its performance, and its security. Several reference implementations of CREHMA have developed and can be downloaded as open-source software. More details about the reference implementations are included in the Appendix B.

## **1.5 Thesis Structure and Research Methodologies**

This cumulative thesis is organized into eight chapters. The core of this thesis are the Chapters 2 to 7. Each of these chapters includes one research paper. This section briefly summarizes the goals and research methodology of these papers.

Paper 1 included in Chapter 2 is a consolidated journal article which is based on four publications ([Gor+14a; LN15a; NL15; NL16]). The goal of this article is to emphasize the need for a general REST-Security framework as well as to address the first two research questions RQ1 and RQ2. The contribution starts by discussing REST-Security specifics as well as demands. It takes the SOAP Web Services security stack (WS-Security) [Nad+06] as a role model and argues that REST-based systems require a similar security stack. To build a REST security stack, this chapter proposes the design of REST-Security components that is based on the same abstraction level as REST itself. As with REST, REST-Security components are defined as a generic guideline including general policies on security-critical assets as well as algorithms for

protecting them. This generic guideline then serves as a methodical framework which can be adopted in any REST-based technologies including HTTP and CoAP. To reinforce the need for a general REST-Security framework, Chapter 2 presents an in-depth related work analysis on 21 available authentication schemes for HTTP and CoAP. The criteria to assess the authentication schemes are based on a threat model and a list of to be protected message elements. The analysis points out the limitation and vulnerabilities of current REST-based security means. For instance, many HTTP signature schemes only protect the request message or do not sign mission-critical header fields. The evaluation also observes that available HTTP and CoAP signature schemes are very diverse, although they follow the same objective. All these findings highlight the need for a more methodical and unified approach, which is mandatory to meet the requirements of a stable and reliable modern software system. As a result, this publication proposes the general authentication scheme REMA, a guideline containing policies and algorithms for ensuring end-to-end authenticity and integrity of REST messages. In the next step, REMA serves as a generic framework for deriving REHMA and RECMA, which provides end-to-end authenticity and integrity for HTTP and CoAP messages. The evaluation of REHMA and RECMA shows that both schemes outperform available HTTP and CoAP signature schemes in terms of security.

Based on the same methodology of implementing REHMA and RECMA, Chapter 2 also discusses the requirements and challenges for developing a REST message confidentiality scheme. Likewise, such a scheme should provide a general guideline containing a policy on to be protected elements as well as algorithms describing how to ensure confidentiality. End-to-end message confidentiality in REST-based systems must be developed under very specific requirements, as REST requires messages to be self-descriptive. The message self-descriptiveness is affected when the full content of the message is encrypted. A REST-compliant message confidentiality scheme, therefore, must protect REST messages from unauthorized disclosure while retaining the self-descriptiveness. Hence, a policy of a REST message confidentiality scheme must define what message parts are accessible and what message parts must be kept secret to what class of intermediate system. With such a policy, intermediate systems such as caches can get write and read access to message elements containing caching information while remaining message parts comprising sensitive information for other purposes remain encrypted. As intermediate systems are vital components for the security and scalability of REST-based systems, the outlook of this chapter requires to study intermediate systems in further work. Such studies aim to understand the characteristics and interference of intermediate systems to security and scalability. This knowledge then allows specifying a well-defined read access policy for the REST message confidentiality scheme.

Chapter 3 provides an empirical analysis of fifteen REST-based service description languages. Service description languages have been established as a popular and vital approach for machine-to-machine communication and automatic code generation. The goal of this evaluation is to study the ability of REST-based service description languages to describe security schemes. As with the state-of-the-art analysis of REST-based authentication schemes, this chapter reveals many limitations. For instance, it highlights that only seven of the fifteen analyzed REST-based service description languages supports the definition of security schemes by default. Moreover, the supported security schemes are mostly constrained to authentication and authorization schemes. None of the approaches support HTTP signature schemes by default. Only one service description language provides an extension that allows describing other protection means. Another important observation is that all analyzed REST-based service description languages are only dedicated to HTTP-based application. None of the approaches consider CoAP. Overall, the chapter emphasizes the huge demand for REST-based API description languages as many

specifications exist. However, due to the detected shortcomings, this work also argues that much research is still required to establish such a language as a reliable approach for describing REST-based security schemes.

Chapter 4 and 5 address the task mentioned in the outlook of Chapter 2 which requires to study intermediate systems to understand their interference to security and scalability. The main goal of both publications is to answer RQ3. Both chapters present a large-scale analysis of web caching. Chapter 4 provides a testing methodology based on the HTTP caching RFC 7234 [FNR14] and an in-depth literature review. This methodology allows deriving a comprehensive test suite containing 415 test cases for analyzing the caching behavior as well as the compliance of the web caching system to RFC 7234. With this test suite, eleven popular and widely-used web caching systems have been systematically analyzed. Chapter 4 focuses on the four web browser caches of Chrome, Firefox and Safari, and Microsoft Edge. Chapter 5 analyzes proxy caches and a CDN. Both studies reveal many non-compliances and inconsistencies, which may lead to malfunctions with severe consequences to security as well as scalability.

The discovered issues of Chapter 4 and 5 build the foundation for Chapter 6. This work reinforces the investigation in web caching issues and aims at getting more insight to answer RQ3. During further research on web caching systems, a new class of web cache poisoning attacks has been found. The Cache-Poisoned Denial of Service (CPDoS) attack affects the availability of web applications by injecting an error page or useless content into the cache. Benign clients who intend to invoke the target resource, receive a defective response instead of the genuine content. As with other web cache poisoning attacks, CPDoS exploits the semantic gap in the handling of request headers. In particular, it sends a request with an erroneous header, which is forwarded by the cache without any issues. On the server-side, this request provokes, however, an error page which is cached and reused by the intermediary to satisfy further requests of other clients. Chapter 6 presents three variations of CPDoS and evaluates the practicability of these attack vectors to real-world systems. The HTTP Header Oversize (HHO) attack takes advantage of the different request header size limits between the cache and origin server. The HTTP Meta Character (HMC) attack is similar to HHO. It exploits the semantic gap in handling meta characters. The HTTP Method Override (HMO) attack utilizes the so-called method overriding headers to initiate a cacheable error page on the origin server. An analysis on request header size limits, meta character handling, and method overriding header support with fifteen web caches, reveals that millions of real-world websites are vulnerable to CPDoS. All findings have been reported to the respective parties. Based on the feedback of the cache vendors, Chapter 6 provides details on the countermeasures which also have been discussed with other researchers.

Caches have a significant impact on the security and scalability of REST-based systems. When used properly, web caching systems provide increased scalability and availability. Caches can, however, also be exploited to massively impair the scalability and availability as shown by CPDoS and many other web caching attacks. The consideration of caches in security schemes is, therefore, of paramount importance in order to be in alignment with the REST principles but also to mitigate cache-related vulnerabilities. Chapter 7 extends REHMA which was proposed in Chapter 2 and introduces Cache-aware REST-ful HTTP Message Scheme (CREHMA). Such an extension is required, since available HTTP signature schemes do not support caches. The omission of caches in available signature schemes forces to disable caching when end-to-end security is required. Content providers can use CREHMA with the same security baseline of REHMA, but with the addition that they can still enjoy the benefits of caching. The contribution

of Chapter 7 starts with augmenting the threat model of Chapter 2 based on the knowledge of the studies in web caching. Moreover, the chapter points out the limitation and vulnerabilities of available HTTP signature schemes related to caching. The extended threat model and related work analysis then build the basis to derive CREHMA. To make CREHMA cache-enabled, Chapter 7 appends cache-related headers to the list of to be protected messages elements defined in Chapter 2. Moreover, the signature and verification algorithm of REHMA have been adapted to consider caches. In the next step, prototype implementations of CREHMA with distinct programming languages have been developed to evaluate the compatibility, performance, and security. The compatibility evaluation shows that CREHMA can be used with all kinds of real-world caches without the need to change the source code of the intermediary. Regarding the performance, Chapter 7 demonstrates that CREHMA barely affects latency in comparison with message exchanges without end-to-end protection. In the security analysis, Chapter 7 additionally highlights that CREHMA thwarts the same attack vectors of REHMA but also can detect web cache poisoning attacks such as HTTP Request Smuggling [Lin+05], Host of Trouble [Che+16] and HTTP Response Splitting [Kle04]. The reference implementation of CREHMA in distinct programming languages are published as free and open-source software, which can be downloaded via GitHub. More details on the reference implementation of CREHMA can be found in the Appendix B.

Chapter 8 concludes this thesis with a reflection on the contribution as well as an outlook on challenges and further work in the security for REST-based ULS systems.

# Chapter 2

## On the Need for a General REST-Security Framework

### Summary of this publication

**Citation** L. Lo Iacono, H. V. Nguyen, and P. L. Gorski. *On the Need for a General REST-Security Framework*. In: *Future Internet* 11.3 (2019). URL: <https://doi.org/10.3390/fi11030056>

**Status of Paper** Published

**Type of Paper** Research Paper (Journal)

**Impact Factor** ResearchGate: 1.12

**Aim** The paper reviews the current activities in REST-based authentication schemes. Based on this analysis, the paper emphasizes the need for a general REST-Security framework.

**Methodology** We define a threat model and a list on to be protected REST message assets. Based on the threat model and the list, we conduct an in-depth empirical analysis of 21 REST-based authentication schemes.

**Contribution** In the empirical analysis, we detect many vulnerabilities and inconsistencies. With this background, the paper contributes a methodical approach on how to establish REST-Security as a general security framework for protecting REST-based service systems of any kind. The first adoptions of the introduced approach are presented in relation to REST message authentication with instantiations for REST-ful HTTP and REST-ful COAP.

**Co-authors' contribution** See Paper 1 in Section [1.1.1](#).



## 2.1 Introduction

Representational State Transfer (REST) [Fie00] is an architectural style for designing distributed services systems that scale at large. This is achieved by a set of defined architectural constraints. REST-based systems have to be, e.g., *stateless* and *cacheable* in order to ensure the propagated scalability. The *uniform interface* is another important constraint, which provides simplicity of interfaces and performance of components' interaction. The benefits coming along by adhering to these constraints are amongst the main driving forces for the increasing adoption of service systems based on REST.

Currently only a limited set of technologies exists, which can serve as a foundation for implementing REST-based systems. HTTP [FR14b] is by far the most dominant choice. This fact is the source for many misinterpretations in which REST is often equated with HTTP. Consequences emerging from this reasoning are manifold. One related to security is the adoption of transport-oriented protection only, as common for conventional Web-based applications by means of TLS [Res18]. This is by far not sufficient as an exclusive safeguard for REST-based services, since they are constrained to be *layered*. Hence, these systems consists of intermediaries, which perform functions on the data path between a source host and destination host, most commonly on the OSI application layer [CB02]. Examples of such intermediate systems include caches, load-balancers, message routers, interceptors and proxies. In order to be able to perform their tasks, intermediate systems need to terminate transport security, which as a result does not reach from end to end. This remains opaque to the user and the obtained security level depends on many more stakeholder than the two endpoints. Durumeric et al. [Dur+17] revealed that many current security interceptors struggle with the implementation of transport-oriented security protocols, as they build intermediate systems that decrease security or even provide implementations that are severely broken. Also, transport-oriented security is not designed to fulfill the security requirements of Ultra Large Scale (ULS) [Fei+06] systems and distributed service-oriented applications in general. The various entities involved in chained processing steps require adopting more fine-grained and message-related security means such as partial encryption and signature as, e.g., provided by the WS-Security [Nad+06] standard for SOAP-based Web Services [Gud+07].

Moreover, different protocols following the REST principles are starting to emerge in domains other than the Web or the Cloud. For implementing IoT services, for instance, CoAP [SHB14] is taking root as REST-compliant protocol. DTLS [RM12], the UDP-based flavor of TLS, is applicable as transport-oriented security measure here likewise. Again, the REST inherent constraint of composing systems out of layers, in many cases prohibits the adoption of transport-oriented security as single line of protection. Especially in the IoT domain, most of the use cases comprise high security demands, asking for more elaborated and pluralistic safeguards.

The limited protection of transport-oriented security in REST-based systems has already been addressed by several research and development approaches as will be discussed thoroughly in Section 2.5. From these, some REST message security technologies have emerged that can be used in conjunction with transport security. Still, these approaches are available for certain technologies only, mainly HTTP and CoAP until now. As REST defines an abstract concept, its implementations are not restricted to these two particular technologies, though. Since REST has been established as an important paradigm for building large-scale distributed systems, more REST-ful protocols are expected to evolve prospectively. The Remote APDU Call Secure (RACS) [Uri19] protocol is one example. It is an emerging REST-ful protocol for accessing

smartcards. Beside transport-oriented security no further protection means have been proposed for the RACS draft standard so far.

From these basic observations, the need for a general REST-Security framework gets apparent. The objective of this paper is to close this gap while providing the following contributions. First, this paper analyses the actual security demands of REST-based systems thoroughly and emphasizes the specific REST characteristics that necessitate a dedicated REST-Security, which needs to be defined at the same abstraction layer as REST itself and independent from any concrete technology in the first place. Then, a comprehensive consolidated review of the current state of the art in respect to REST-Security is provided. Finally, the paper contributes a general REST-Security framework alongside with a methodology on how to instantiate it for a particular REST-conformance technology stack in order to facilitate the protection of REST-based service systems of any kind by consistent and comprehensive protection means. Available as well as upcoming REST-ful technologies will benefit from the introduced methodology and the proposed general REST-Security framework at its core.

For this purpose, the remaining of this paper is organized as follows. The foundations in respect to the architectural style REST are laid in Section 2.2. The methodology for deriving the envisioned general REST-Security framework is laid in Section 2.3. The subsequent sections follow this methodology accordingly, starting with capturing the demand in terms of required service security technologies in Section 2.4. Due to the lack of a widespread adoption of REST other than the Web—but without the loss of generality—the security demands and specific requirements are analyzed based on the Web Services security stack. In Section 2.5 the related work and current practice is presented and assessed. A general security framework that reflects the particular characteristics and properties of REST is introduced in Section 2.6. Based on this approach, Section 2.7 proposes an adoption of the framework to two prevalent concrete REST-based protocols, HTTP and CoAP. An experimental evaluation of these schemes against the related work based on prototype test-beds is given in Section 7.7. The paper concludes in Section 2.9 and provides a brief discussion on future research and development demands.

## 2.2 REST Foundations

Besides the dissertation of Roy Fielding [Fie00] there does neither exist a definition nor a unified understanding of the term REST and its underlying principles and concepts. Often enough it is mistaken as being a standard composed of its underlying foundations HTTP and URI [BFM05]. The source for this diffuse view on REST lies mainly in the fact that the two aforementioned standards have been the only notable technology choice for implementing REST-based service systems ever since. For the purpose of this paper it is henceforth demanding that the term REST is defined unambiguously.

The aim of REST is to provide a guideline for designing distributed systems that possess certain traits including performance, scalability, and simplicity. These architectural properties are realized by applying specific constraints to components, interfaces, and data elements. These constraints are subsequently introduced with the guidance of Figure 2.1.

REST is constrained to the *client-server* model in conjunction with the request-response communication flow. A REST client performs some kind of action on a targeted resource by issuing a request. For this, the request must contain a resource identifier and the action to apply to the addressed resource. Depending on the action, the request and response messages may contain



additional meta data elements, which are categorized in resource data, resource meta data, representation data, representation meta data, and control data. The set of available actions in conjunction with a unique scheme for identifying resources as well as the additional meta data is known as the *uniform interface* since it is consistent for all managed and provided resources.

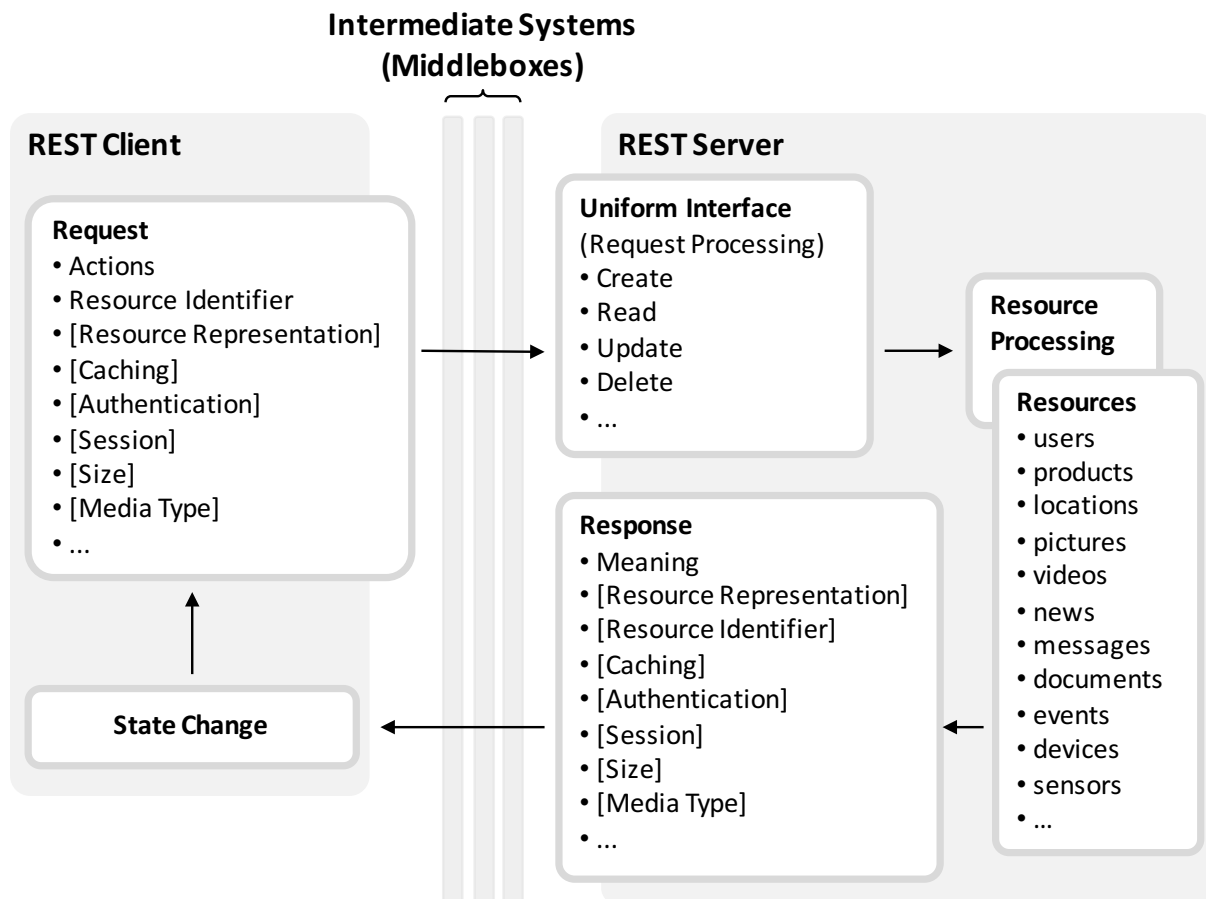


Figure 2.1: Overview of the REST constraints and principles (on the basis of [Gor+14a])

Since REST-based systems are constrained to be *stateless*, messages need to contain all required data elements in order to relieve the server from maintaining state for each client. As REST messages embody all required data elements, which are predefined and standardized by the uniform interface, their semantics are visible and are hence *self-descriptive* for all intermediaries and endpoints so that all components in a REST architecture can understand the intention of a message without knowing each other in advance. In a request to read access a resource, for instance, the request contains the resource identifier along with representation meta data to signal in what data format the resource should be delivered from the server to the client. Moreover, a request can include further meta data required by intermediaries including state and caching information. The according response provides information on its meaning and in case it denotes that the addressed resource is available, it is contained in the message body in the requested representation. Once a response is received, it transfers the receiving client into a new state. In another setting, in which the request triggers the creation of a new resource, the request contains the resource in the request message body in some representation. The response then gives feedback on the resource state and whether it has successfully created or not. Again, further meta data elements can be included in addition, providing information on the authentication, the session and the freshness of a resource in respect to caching. Moreover, the

meta data elements as well as the resource representation of the response may contain further resource identifier—i.e. hyperlinks. Based on these resource identifiers and their description, a client is able explore other resources and transfer its state by starting a new request with a distinct resource identifier and meta information. This REST property is known as *Hypermedia As The Engine Of Application State* (HATEOAS).

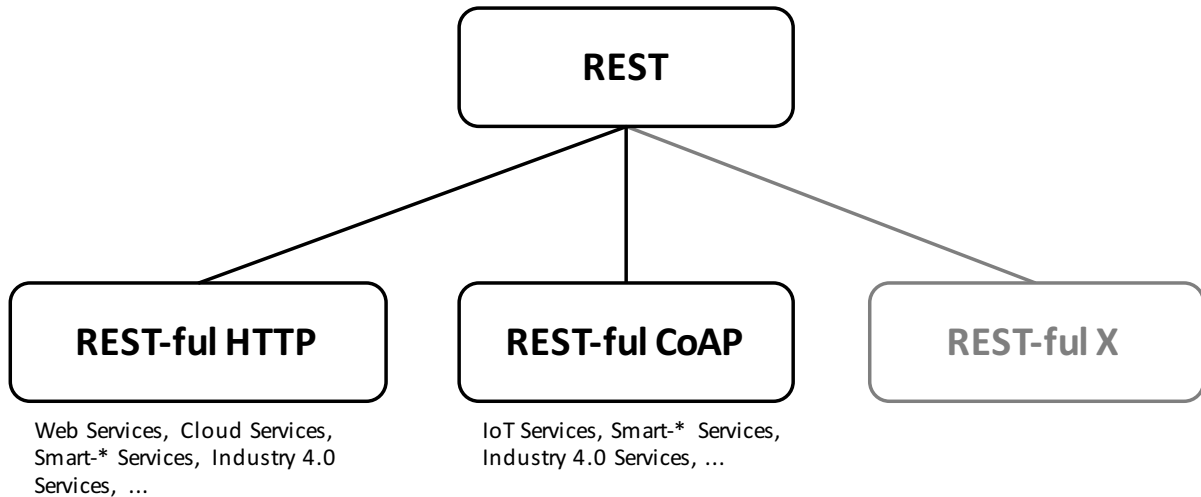


Figure 2.2: Instantiation of the general REST architecture style to specific REST-ful protocols

The principles and constraints representing REST are fairly abstract making it adoptable in any environment that contains technologies suitable for implementing the REST constraints. This coherence is illustrated by Figure 2.2. HTTP is one protocol that is in conformance with the REST constraints and principles as it is based on the client-server model and the interaction is stateless. Moreover, it specifies a uniform interface, which specifies a set of predefined request actions, i.e. the HTTP methods, and a set of additional meta data for transferring different resource representation or controlling the cache behavior for example. Additionally, HTTP uses a resource identifier syntax, i.e. the URI standard [BFM05], for addressing resources. An instantiation on the technological basis of HTTP results in REST-ful HTTP [LN15b], the foundation for building REST-based Web, Micro or Cloud services, which in turn are used to build Smart-\* and Industry 4.0 applications. More specifically, the fifth generation of mobile communication systems (5G), e.g., adopts REST-ful HTTP for implementing a Service-based Architecture (SBA) providing core network functions as REST-ful services [Mao+]. Another evolving application domain of REST can be found in the Internet of Things (IoT) [Car+18]. Here, the REST-conformance Constraint Application Protocol (CoAP) [SHB14] is used to implement distributed service systems consisting of a large number of resource-restricted nodes [BCS12]. CoAP adopts most of the HTTP characteristics. It utilizes the same request actions and the URI standard for specifying the uniform interface. Also, CoAP defines similar meta data for transferring and controlling the cache behavior. The main difference between CoAP and HTTP lies in the fact that CoAP is a binary protocol, whereas HTTP is text-based. Other technical instantiations of REST are equally possible and might appear in the future such as the Remote APDU Call Secure (RACS) [Uri19] protocol, which is still being standardized. This abstraction hierarchy is an important fact to consider carefully when researching on REST or REST-Security.

## 2.3 Methodology

To derive a general framework for REST-Security, the methodology depicted in Figure 2.3 has been applied. In a first phase (see Section 2.4 for details), the specific needs of REST-based systems in terms of security have been derived by analyzing available standards and academic work in related domains and contrasting them with characteristics of REST-compliant systems. Moreover, a common and realistic threat model is defined and used as a basis for the subsequent phases.

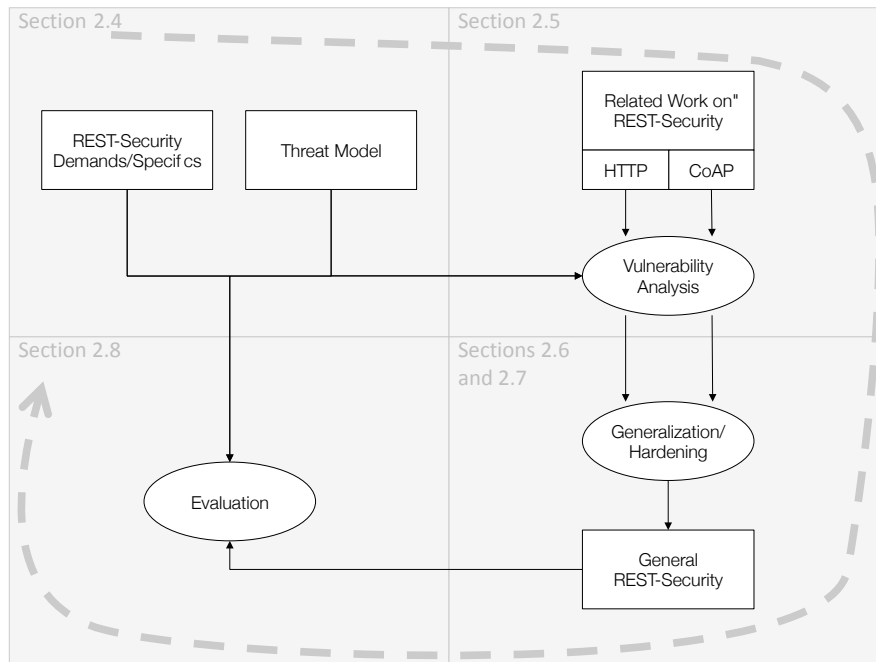


Figure 2.3: Adopted methodology to derive a general framework for REST-Security

To obtain an in-depth understanding on how REST messages are protected by available means, a comprehensive study of schemes introduced in literature as well as deployed in practice has been executed (see Section 2.5 for details). 21 approaches have been identified in total and all of them have been evaluated in respect to the specific security demands of REST-based systems and the determined threat model.

As none of the analyzed REST message security schemes fulfills all necessary requirements and is free of vulnerabilities in the given threat model, a new approach to REST-Security has been developed in an adjacent activity (see Sections 2.6 and 2.7). Governed by the main outcomes of the previous studies, the generalization as well as hardening of the proposed schemes have been the goal. To be able to get a proof-of-concept, particular entities of the general REST-Security framework have been instantiated. As most of the available related work is focusing on REST message authentication, the implemented instantiations of our framework do so as well for HTTP and CoAP.

To evaluate the derived and introduced general REST-Security framework and more specifically its particular instantiations have been examined in experimental test-beds using prototypes (see Section 7.7 for details). For this purpose, implementations of the related schemes—as far as

openly accessible—have been integrated in experimental test environments. For comprehensibility reasons the source codes of the introduced scheme REHMA and RECMA have been published and made available in the public domain.

## 2.4 REST-Security Demands and Specifics

When considering REST for the design of service systems of any kind, the general security demands of SOA [KC08] apply. The ability of REST-based systems to also comply with the SOA principles has been analyzed and shown in [Gor+14b]. As SOAP-based Web Services have been and still are a dominant technology stack for implementing SOA-based systems, the evolved security stack for SOAP-based Web Services can serve as reference [Gor+14a].

### 2.4.1 SOAP-based Web Services Security Stack

The SOAP-based Web Services technology stack includes an extensive set of security standards (see Figure 2.4) [Gor+14a].

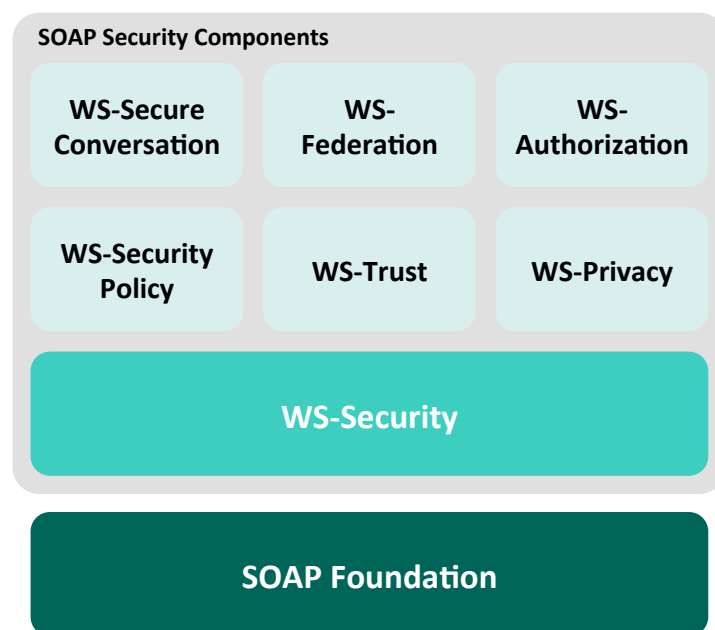


Figure 2.4: Security stack for SOAP-based Web Services [Gor+14a]

SOAP uses XML [Bra+08] as platform-independent and extensible data description language for defining the structure and semantics of the protocol messages. To ensure basic security services for SOAP messages such as confidentiality and integrity, the WS-Security [Nad+06] specification has been standardized, which is based on XML Encryption [Ima+13] and XML Signature [Bar+08]. Upon these foundations, further reaching security concepts are provided. The fundamental condition for any security systems is trust. WS-Trust [Nad+07] introduces a standard based upon WS-Security for establishing and broking trust relationships between service endpoints. WS-Federation [GN09] extends WS-Trust in order to federate heterogeneous security realms. It provides authorization management across organizational and trust boundaries. The authorization management within those realms is described in WS-Authorization. Privacy

constraints are covered by the WS-Privacy specification. It allows handling privacy preferences and policies between client and server. Secure communication, trust, federation, authorization and privacy need a mechanism to negotiate and handle security policies. WS-SecurityPolicy [Nad+12] specifies how constraints and requirements in terms of security are defined for SOAP messages. It is a framework, which allows Web Services to express their security demands as a set of so-called policy assertions. WS-SecureConversation [Nad+09] expands the security mechanisms for a conversation between two communication partners. This OASIS standard defines how a secure exchange of multiple messages has to be established in terms of a session [RR04].

## 2.4.2 REST-ful Services Security Stack

REST-based services require a comparable set of technologies in order to enable developers to implement message-oriented security mechanisms as required by the surrounding application context.

The currently available security stack is, however, rather scarce in comparison to the SOAP-based Web Services security stack (see Figure 2.5) [Gor+14a].

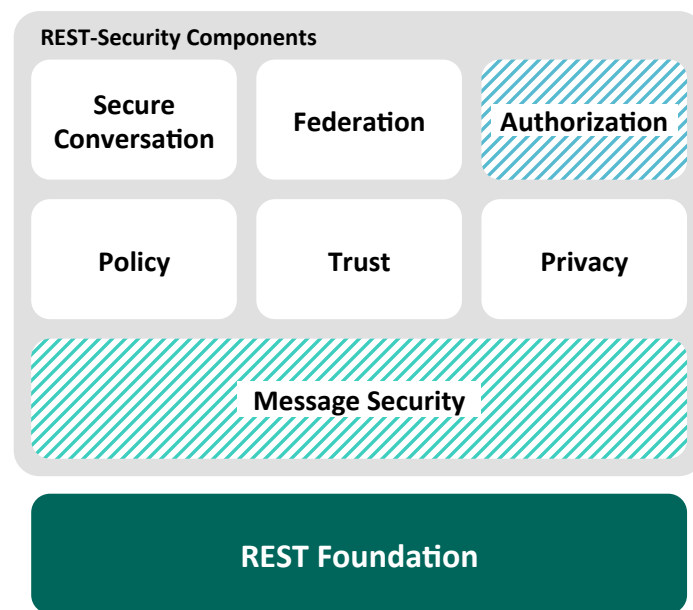


Figure 2.5: Desired security stack for REST-based Web Services [Gor+14a]

Even the fundamental message security layer is not available completely (visualized by the dashed area) [Gor+14a; LN15a]. Some standards related to the authorization of service invocations such as OAuth [Har12] and drafts on identity federation [Hed+18] are at hand, but the rest of the higher order security concepts including trust, secure conversation and so forth are lacking entirely. Still, the depicted security stack for REST-based services is a necessity and thus needs to be developed.

### 2.4.3 REST-Security Specifics

Although both security stacks have their similarities, the plain adoption of WS-Security to REST and instantiations of REST is not feasible in a straightaway manner. The specifics of REST have to be considered carefully in order to obtain a suitable and seamless security for REST-based services.

What needs to be taken into account first is the abstraction layer of REST and its instantiations. REST itself is a very general concept and needs to be handled accordingly. Thus, a simple mapping of the concrete WS-Security technologies to construct REST-Security is not feasible, since both reside on different abstraction layers. Since REST represents an abstract model, security components for this architectural style need to be considered and defined on the same abstraction layer as well. Consequently, REST-Security needs to be a general framework composed of definitions, structures and rules on how to protect REST-based systems. The term general in this context has to be understood as generic in the sense that the schemes contained in the REST-Security framework are not bound to a specific REST-based technology or protocol only, but are applicable to any REST-ful technology. Such a general REST-Security framework would then support a guided adoption and implementation to any concrete REST-ful protocol (see Figure 2.6).

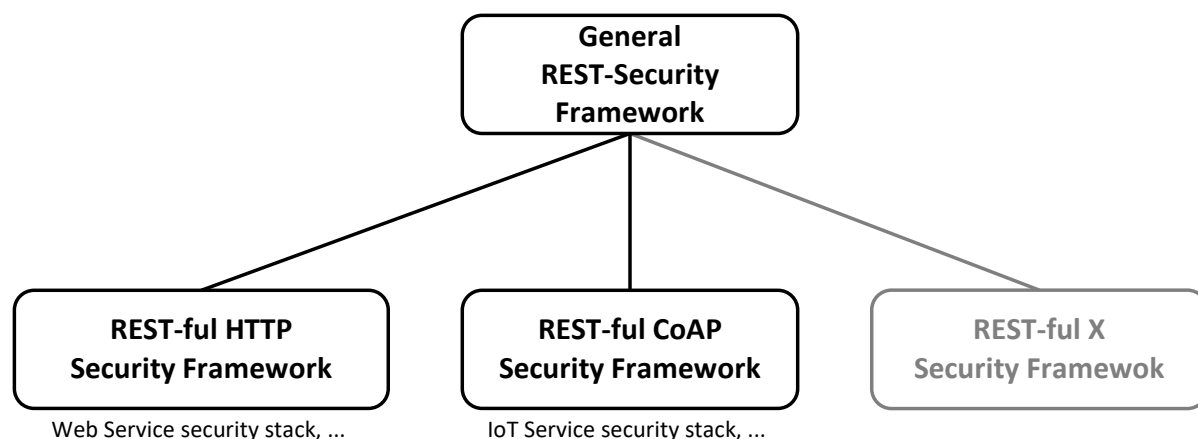


Figure 2.6: Instantiation of the general REST-Security framework to specific REST-ful protocols

Another REST specific is that there is no self-contained REST message, but the relevant data items are scattered around the service protocol and the service payload (see Figure 2.7). SOAP messages, in contrast, are a self-contained XML structure. Both, the meta data as well as the payload in form of a service operation or its corresponding result are enclosed in one XML document. Thus, with the application of security mechanisms based on the technologies shown in Figure 2.4, both message parts can be covered. This is, however, not the case for REST messages. Referring again to REST-ful HTTP as an example, the meta data is included in the HTTP header, whereas the resource representation is inside the HTTP body. Since both parts are disjoint for many reasons, distinct security mechanisms need to be applied in a balanced manner. If this is not being recognized, novel vulnerabilities might be exploitable in the future.

Table 2.1 shows a set of possible attack vectors, which can be applied to REST-ful HTTP messages that do carry a protected body only. The assumed attacker model is a common man-in-the-middle (MITM) attack, in which an intruder is able to tamper the whole HTTP request

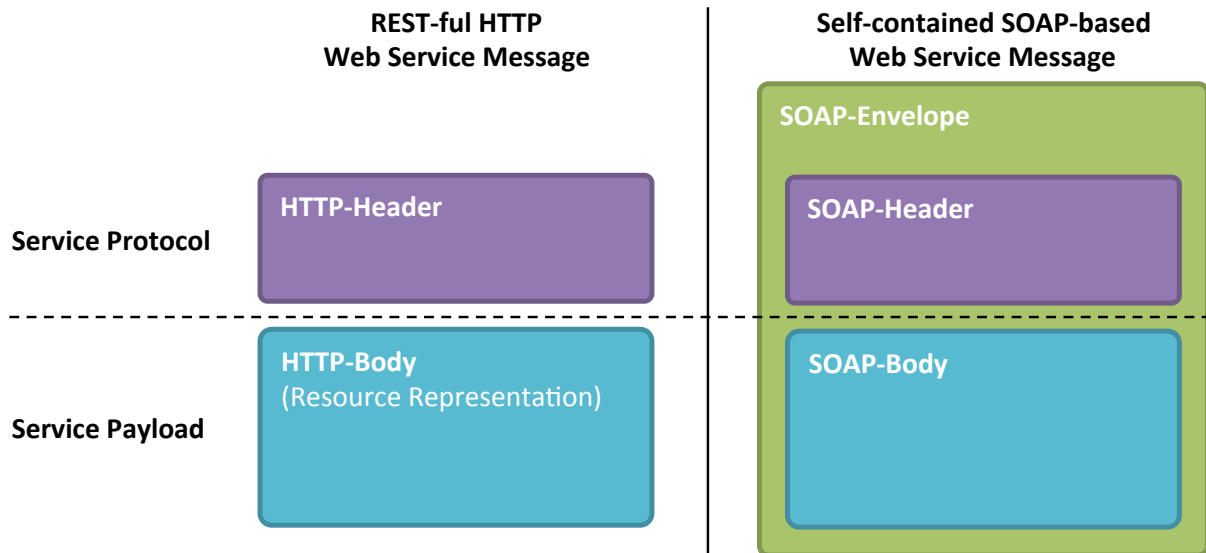


Figure 2.7: Comparison of the SOAP-based service message structure with a REST service message structure exemplified by a REST-ful HTTP instantiation

and response messages due to exploited transport security vulnerabilities or a compromised intermediate system. Attack #1 is based on a GET request that does not contain a resource

#	Original REST-ful HTTP message	Tampered REST-ful HTTP message
1	GET /resources HTTP/1.1 Host: example.org Accept: application/json Content-Length: 0 Connection: keep-alive	<b>DELETE</b> /resources HTTP/1.1 Host: example.org Accept: application/json Content-Length: 0 Connection: keep-alive
2	GET /resources HTTP/1.1 Host: example.org Accept: application/json Content-Length: 0 Connection: keep-alive	GET <b>/evilresources</b> HTTP/1.1 Host: <b>attacker.org</b> Accept: application/json Content-Length: 0 Connection: keep-alive
3	PUT /resources/3 HTTP/1.1 Host: example.org Content-Length: 100 Content-Type: application/jose+json Connection: keep-alive  <protected body>	<b>DELETE</b> /resources/3 HTTP/1.1 Host: example.org Content-Length: <b>0</b> Content-Type: application/jose+json Connection: keep-alive  <protected body>
4	POST /resources HTTP/1.1 Host: example.org Content-Length: 100 Content-Type: application/jose+json Connection: keep-alive  <protected body>	POST /resources HTTP/1.1 Host: example.org Content-Length: <b>120</b> Content-Type: application/jose+json Connection: keep-alive  <b>&lt;replaced malicious protected body&gt;</b>
5	HTTP/1.1 201 Created Content-Length: 0 Connection: keep-alive Location: http://example.org/resources/4	HTTP/1.1 201 Created Content-Length: 0 Connection: keep-alive Location: <b>http://attacker.org/resources/4</b>
6	HTTP/1.1 200 OK Connection: keep-alive Content-Length: 100 Content-Type: application/xml  <protected body>	HTTP/1.1 200 OK Connection: keep-alive Content-Length: 100 Content-Type: application/xml <b>Cache-Control: max-age=7200</b>  <protected body>

Table 2.1: Possible attack vectors on unauthenticated REST-ful HTTP messages

representation. Thus, the whole HTTP message remains unprotected providing the surface for a



malicious twist of the GET method by, e.g., the DELETE method. The second attack tampers the resource path and the Host header with the aim of redirecting a client to the attacker's resource. The attack in row 3 emphasizes that even if a message includes a protected body, an attacker is still able to spoof the unsecured header. In the provided case, the attacker can manipulate the HTTP method and the Content-Length header in order to construct a valid DELETE request. Moreover, a malicious replacement of a resource representation is also feasible as shown in row 4. Here, an adversary can substitute a resource representation with its own resource representation. Similar attacks are also possible on responses. Row 5 depicts an example, where the Location header is changed in order to forward a client to a malicious resource. The attack in row 6 presents a deception of the cache behavior. This manipulation misleads the client or proxy to save the response for two hours. As a consequence, any further requests in the next two hours to this resource will be replied by the cache and not by the origin server so that the client can no longer notice a change of the actual resource state. These possible attack vectors can be transferred analogously to REST-ful CoAP messages that are described in [NL15]. Note, that Table 2.1 lists a set of potential attacks vectors, which the authors identified and considered critical. This is not an exhaustive list yet. Future work may uncover additional attack vectors that will provide more arguments for appropriate message-level safeguards.

## 2.5 Related Work Analysis

The argued need for a general REST-Security framework is further examined by an analysis of the current practice and the available research. The analysis captures the correct security mechanisms and evaluates them according to the specific of REST-based systems and the attacker model given in the previous section. The related work has evolved so far in a relevant manner on REST-ful HTTP and REST-ful CoAP only. Moreover, most of the available work has been conducted in relation to basic service message security with a focus on authentication and authorization. Thus, the subsequent analyses are driven by these prerequisites [LN15a].

Note, that in comparison to [PCA16] the related work analysis focuses on approaches protecting the specifics of the uniform interface of REST-based systems in general. Security techniques targeting a specific application domain are not considered. That is, protection means referring to vulnerabilities of conventional Web applications such as cross-site scripting (XSS), cross-site request forgery (CSRF) or SQL injection are therefore out of scope.

### 2.5.1 HTTP Basic and HTTP Digest Authentication

HTTP Basic [Res15] and HTTP Digest Authentication [SAB15b] have been the two first standards for authenticating HTTP requests. Both schemes require a username and a password for the authentication process. If a client tries to access a resource, which is protected by one of these mechanisms, the server returns an error response message including the WWW-Authenticate header containing the name of the mechanism, i.e. Basic or Digest, and a realm which is a description of the secured resource. In case of Basic Authentication, the client must authenticate itself by sending the former request with an Authorization header, which includes the base64-encoded username and the password. The, i.e. plain-text transfer of username and password transfer is the main downside of this approach. To protect this sensitive information in transit, TLS must be applied additionally. If transport-oriented security is not available, the



hole message remains unprotected and password eavesdropping as well as any kind of request manipulations including the ones in Table 2.1 are feasible.

HTTP Digest Authentication provides a slightly improved approach, as it does not transfer the credentials in clear-text. Here, the username and the password are hashed. Besides the username and password the hash computation includes the URL path, the HTTP method, a client- as well as a server-generated nonce, a sequence number and optionally a quality of protection description. Since this scheme considers the HTTP method and the URL path in the hash calculation, a manipulation of these request message elements is not feasible. However, an attacker can still perform malicious changes of other message entities such as distinct headers and the body.

The other main drawback of both authentication mechanisms is that the request can be authenticated only. Servers are not able to authenticate their responses opening the door for MITM attacks.

### 2.5.2 API-key

API-keys are randomly generated strings, which are negotiated out-of-band between client and server. An API-key is added to the URL or header of every request. According to an analysis of the Web API directory *ProgrammableWeb*, API-keys are currently the most used authentication mechanism in REST-based Web Services [NTL17].

API-keys share the same drawbacks as HTTP Basic Authentication. The API-Key is transferred to the server in plain-text. Thus, the credentials are only protected during transit if transport-oriented security means such as TLS are being used.

### 2.5.3 HOBA

The experimental RFC HTTP Origin-Bound Authentication (HOBA) [FHT15] is a challenge response HTTP authentication method based on digital signatures. If a distinct resource is protected by HOBA and accessed without authentication, the server returns an error response including the WWW-Authenticate header. This header contains a challenge string, an expiration date and an optional realm. To access this protected resource, the client needs to create a signature covering a client-side generated nonce, the base URL of the request, the signature algorithm name, the optional realm, the key identifier and the challenge string. The resulting signature value must then be included in the Authorization header together with the key identifier, the challenge string and the nonce.

HOBA does not ensure the integrity of HTTP requests. To do so, each data transfer in HOBA must be protected by TLS. If transport-oriented security is not present, any malicious change of the request can be performed. As with HTTP Basic, Digest and API-keys, Authentication, HOBA considers the authentication of requests only and does not provide the option to protect responses.

## 2.5.4 HTTP SCRAM

Salted Challenge Response HTTP Authentication Mechanism (HTTP SCRAM) [Mel16] is another experimental RFC. The authentication process is structured in two steps. In the first one, the client sends a request containing an HMAC signature algorithm name, the username and a self-generated nonce. The server replies with the client-generated nonce concatenated with a server-generated nonce, a salt and a sequence number. Based on this information, the client performs the second authentication step. It computes an HMAC signature composed of the password, the salt and the sequence number. To access the HTTP SCRAM protected resource, the calculated signature value is embedded in the Authorization header including the client-generated nonce concatenated with the server-generated nonce and an HTTP SCRAM-specific description. Once the server receives this request, it verifies the signature and the concatenated nonces. If both values pass the verification process, the server returns the desired response to the client. The response contains a signature value as well which is created by means of the client's password, the salt and the sequence number, so that the client can proof the authenticity of the responding server.

Unlike HOBA, API-keys as well as HTTP Basic and Digest, HTTP SCRAM provides the option to authenticate requests as well as responses. However, this approach does not guarantee the integrity of the whole message, as the signature does not cover the body and most of the headers.

## 2.5.5 Mutual Authentication Protocol for HTTP

The experimental RFC *Mutual Authentication Protocol for HTTP* [Oiw+17a] is an approach for authenticating requests and responses without sending the user's password in plain-text. To transfer the password in a confidential manner, the client as well as the server generates a key exchange value each. The generated exchange value of the client is sent via a request to server and the generated exchange value of the server is returned to client by a response.

Based on these client- and server-generated key exchange values, a session secret is calculated. The client as well as the server use this session secret to create a verification value. Included within the Authorization request header of the request or the Authentication-Info response header, the verification value serves as a parameter for the server and the client respectively for validating the authenticity of the communication partner's received messages.

The procedure for computing the verification value, the key exchange value and the session secret is not specified in [Oiw+17a]. A description of algorithms for computing the credentials is provided in a separate specification [Oiw+17b]. Here, the key exchange values are randomly generated. The session secret is a SHA-256 or SHA-512 hash calculated from the key exchange values of the client and the server as well as the user's password. Alternatively, the session secret can be calculated via elliptic curve digital signatures, which integrates the key exchange values and the password in the computation process as well. Both verification values are hashes or digital signatures based on the key exchange values and the session secret.

As the name implies, this approach provide a mutual authentication protocol for clients and servers to verify the authenticity of requests and responses. However, only authenticity can be ensured by this specification. Similar to HOBA and HTTP SCRAM, neither the client nor the server can validate whether other headers or the message body has been manipulated.

### 2.5.6 De Backere et al.

De Backere et al. [De +14] present security mechanisms for REST-based Web Services focusing on mobile clients. Their protection scheme requires the client to authenticate with the username and the password before retrieving any resources. If the authentication process is successful, the server returns a symmetric key as well as a token representing the key identifier and a timestamp for avoiding replay attacks. Based on these three credentials, a client can send authenticated requests. This is realized by embedding the token and timestamp within the request. The next step signs the request body with the symmetric key. Optionally, the same symmetric key can also be used for encrypting the request body. To protect the response, the server can utilize the generated symmetric key for authenticating and encrypting the body of the responses. Alternatively, the approach of De Backere et al. provides the option to sign the response body with the server's private key.

The advantage of this approach is the consideration of authenticity, integrity and confidentiality of HTTP requests and responses. However, only the message body is protected by this scheme. The header is left unprotected. Another drawback is a missing description defining whether the token, the timestamp and the computed signature value must be included in the header or body.

### 2.5.7 Peng et al.

Peng et al. [PLH09] present an academic approach which is based on HTTP Basic and HTTP Digest Authentication (see Section 2.5.1). This scheme requires the client to compute two hashes, which are then added to the HTTP header. The first hash is calculated on the basis of a server-generated nonce, a timestamp and a password. The second one is a hash of the username, the realm, the server-generated as well as client-generated nonce, the sequence number, the corresponding HTTP method and the URL path. Both computed hashes including the nonces, the timestamp, the sequence number and the realm are stored in new defined headers before sending the message to the server.

The authentication mechanism of Peng et al. only considers the HTTP method and the URL path in the hash calculations. Other header entries and the body are not secured. Moreover, the approach offers neither an authentication nor an integrity protection of the response.

### 2.5.8 FOAF+SSL/WebID

The Friend-of-a-Friend project (FOAF) [BM14] project aims to define a specification for linking people and information on the Web. In FOAF people, agents, groups and their relations can be described in a machine-readable manner. FOAF+SSL [Sto+09], also known as WebID [SH13], extends FOAF by authentication. The trust model of FOAF+SSL is based on the Web of Trust (WOT) [KR97] where each entity acts as a trusted third party. Each WebID certificate contains a link to a corresponding FOAF description, in which an entity and its relations to other entities are defined. Based on this description and references a WOT can be built. As the name FOAF+SSL implies, the WebID certificate is used to establish a TLS connection likewise. Doing so, authenticity, confidentiality and integrity can be ensured in the transport layer.

FOAF+SSL does not provide any safeguards for the application layer. The security is based on TLS and WOT only. Thus, systems using FOAF+SSL for authentication are still vulnerable for the attacks described in Table 2.1.

### 2.5.9 Google, Hewlett Packard and Microsoft

The cloud storage services of Google [Goo17], Hewlett Packard (HP) [Hew14] and Microsoft [Mic17] utilize an enhanced API-keys mechanism that prevents eavesdropping the key in transit. Instead of simply including the API-key directly to the URL or HTTP header, clients signs the request. Conceptually, the core signing process of all three operating cloud storage services is equal. A string to be signed is constructed by concatenating the HTTP method with the resource path including the query (unless HP, which makes use of the resource path only) and a fixed set of headers. Independent of the exact composition of these sets, only the timestamp entry is mandatory. All other specified headers—including for instance the Content-Type or Content-MD5 entries—are optional. The concatenated string is signed by the API-key. The signature value is enriched with further signature-related meta data such as the signature algorithm name and a key identifier. This generated authentication information is finally inserted into the Authorization header. Google supports an alternative option, which allows incorporating the authentication information inside the query part of the URL.

The defined sets of headers to be considered by each of these provider-proprietary approaches do not consider all security-relevant message elements (see Table 2.1). Missing entries include, for instance, the Host and the Connection header. These omissions enable an adversary, e.g., to redirect the message to another system or to manipulate the connection management. Moreover, the providers do not stringently require considering a hash of the body in signature computation. Clients may create the Content-MD5 header to integrate a hash of the body in the signature, but they do not have to. Integrating a hash value covering the body's resource representation into the string to be signed is a vital requirement in order to provide the integrity of the whole REST message. Ignoring this opens the door for spoofing the resource representation. The last but not least observed issue is the lack of mutual authentication, due to leaving the response out of the protection sphere. Thus, a client cannot proof the authenticity of a response providing the surface for MITM attacks.

### 2.5.10 Amazon

Another provider-proprietary approach deployed by the Amazon Simple Storage Service (S3) requires service invocations over HTTP by to be signed [Ama19b]. As with the other three commercial cloud storage services, S3 concatenates the HTTP method, the URL's resource path including the query and a set of headers to a string that is to be signed. The authentication approach of Amazon offers, however, more flexibility as it allows protecting application-specific headers. This is realized by a list that specifies the headers required to be appended before signing or verifying the HTTP message. When this list is used, the request must contain at least the Host header, a header containing a timestamp and the x-amz-sha256 header, which stores a SHA-256 hash of the body. The list is then stored together with the signature value and the remaining authentication information either within the Authorization header or in the URL. Based on this list, the S3 service checks what headers are covered by the signature. If one of the required headers is not contained in the list, the service rejects the request.

The benefit of Amazon's approach is the required hash of the body in the signature generation. Amazon sets, however, the Host header, the timestamp and the x-amz-sha256 header as mandatory only. Consequently, further important meta information such as the Content-Length, the Content-Type and the Connection header are not considered. Thus, an attacker is able to manipulate the resource representation and the connection, if these headers have not been signed. With the aid of the list, an adversary can extract what has been signed and what not. If the Content-Length and Content-Type header are not in the list, a replacement of the resource representation with another resource representation with the same hash value is feasible. Taking the two aforementioned headers into account is crucial to mitigate such attacks. By this, the attacker has to find a resource representation that has the same hash value, size and media type as the actual body. Also Amazon's HTTP authentication scheme suffers from not taking the response into account.

### 2.5.11 Signing HTTP Messages

A standard dealing with the authentication of HTTP messages is the *Signing HTTP Messages* draft of the IETF [CS19]. Similar to the discussed proprietary approaches, a signer has to concatenate the HTTP method, the resource path including the query and a set of headers to a string to be signed. The concatenation order of the headers is determined by the signer, which creates a corresponding list. This list is embedded in the Authorization or the newly defined Signature header together with the signature algorithm name, the key identifier and signature value. Using this list, however, is not required. An absent list results in considering the Date header in the signature generation only. Consequently, a present list must contain at least a Date entry.

Besides this header, the proposal does not consider additional meta data relevant to ensure HTTP message authentication. The client can optionally add more header entries to the signature string if required and aware of the consequences of a too narrow protection sphere. Furthermore, the draft does not require incorporating a hash of the body in the signature computation. Moreover, it does not make clear, how a server needs to authenticate a response. Signing the response is mentioned at the beginning of the draft, but in the rest of the specification it is not elaborated any further.

### 2.5.12 OAuth

OAuth [Ham10; Har12] is an authorization framework for granting access to end users' resources for third party applications. Currently, two versions of OAuth have been published.

The OAuth v1 specification of the IETF [Ham10] has an inherent support for protecting a request by a signature. The signature string is the concatenation of the HTTP method, the resource path including the query, the Host header and a set of OAuth v1 specific parameters. The latter parameters consist of a realm, a key identifier that is called consumer key, an OAuth token, a timestamp and a nonce. OAuth v1 does not enable to add any other parameters or headers in the signature. The authentication information is stored in the Authorization header. Like the other approaches discussed so far, the authenticity of the request is considered solely. No means for signing a response have been defined.

In contrast to the first version, OAuth v2 does not include any security means on its own [Har12]. Instead, the security is merely based on TLS. If a message-oriented protection is yet required, OAuth v2 can be augmented by either the *OAuth MAC Tokens* [RMT14] specification or by the extension *A Method for Signing an HTTP Requests for OAuth* [RBT16]. The OAuth MAC Tokens draft demands to sign the HTTP method, the resource path including the query and at least the Host header. Further meta data can be considered by defining a list similar to some of the previously discussed approaches. The resulting signature value has to be included into the Authorization header.

The second OAuth v2 extension *A Method for Signing an HTTP Requests for OAuth* uses JSON Web Signature (JWS) [JBS15] to guarantee the authenticity of HTTP messages. The JWS object used in this specification owns a set of members, which contains the method, the host including the port, the resource path, the query, the headers, an HMAC authenticator of the body and a timestamp. Using JWS as the pillar can be a stable groundwork, since it is a well advanced IETF draft for signing JSON objects [Bra17] that is already used in many applications. However, the main drawback of this specification lies in the fact that all mentioned JSON members are optional. Even though most of these elements are vital to guarantee the authenticity and integrity of an HTTP message, none of them is set as mandatory for the signature. Also, this draft does not state any information whether the JWS object is stored in a header or in the body.

The common problem of both OAuth versions is the tight coupling to the actual application domain of these authorization frameworks. As a result, adopting these standards to other contexts is not feasible in a straightforward manner. As with the other approaches, the major disadvantage of the OAuth protocols is that they do not specify a protection of the response.

### 2.5.13 Serme et al.

Serme et al. [Ser+12] introduce the first approach addressing the protection of HTTP responses by proposing a REST-ful HTTP message authentication protocol, which protects the request as well as the response. Their approach introduces new headers containing the certificate ID, the hash algorithm and the signature algorithm name. The input to the signature algorithm is a concatenation of the body, the URL, the hash algorithm name, the signature algorithm name, the certificate ID and a set of headers forming a string to be signed. The generated hash and signature values are stored in separated, newly defined headers each. Moreover, Serme et al. propose an encryption and decryption scheme for HTTP messages.

One drawback of [Ser+12] is the missing reference implementation. This paper provides two pseudo code notations of the signature generation and verification schemes as well as another two of the encryption and decryption schemes. These algorithms do not clearly state whether a timestamp or the HTTP method are considered in the processing. Moreover, they do not specify any order of the concatenation or some form of policy, which retains the order. Likewise, the approach does not define what headers need to be obligatory protected. That is, it is not clear whether all headers or a subset of them must be signed/encrypted.

### 2.5.14 Lee et al.

Lee et al. [LJK15] [LJK17] define a method for signing and encrypting HTTP messages. The key pairs for performing the encryption, decryption and digital signatures are generated by a



third party entity. Before a client and a server are starting to communicate with each other, both parties must request two public points (p1, p2), representing the master public key, and a private point (sQ) from a Private Key Generator (PKG). The PKG is the trusted third party service. Its only task is to send the master public key and the private point.

Based on this master public key and the other endpoint's URL, the client and the server can compute the public key of their communication partner. The private point sQ is then used by both parties to calculate their own private key respectively. Once the key pairs and the public key of the counterpart is present, the client and the server can use the communication partner's public key to encrypt the HTTP message. Before starting the data transfer, the encrypted message is signed by the private key of the corresponding endpoint.

The approach of Lee et al. ensures the authenticity, integrity and confidentiality of the whole HTTP message. Moreover, requests as well as responses are protected by this scheme. However, the encryption of the whole message with the aim that only the endpoints are able to decrypt and interpret, violates the self-descriptive constraint of REST messages [Fie00]. That is, only the client and the server can understand the intention of the message. Intermediate systems are not able to process the fully encrypted and signed message, as they possess neither the corresponding private key of the client nor the server. If an intermediary is not able to understand and process a traversing message, it may reject forwarding the message or cancel the communication. Hence, ensuring the confidentiality of REST messages requires to cope with special challenges in order to be in conformance with the REST principles. Requirements for defining a confidentiality scheme in REST are discussed in Section 2.6.3. Another shortcoming of this approach is a missing time variant parameter in the signature process, which makes the scheme vulnerable to replay attacks.

### 2.5.15 OSCORE

Object Security for CoAP (OSCORE) [Sel+18] is a draft standard providing encryption, integrity and replay attack protection for CoAP messages. The CoAP message payload and a set of security-relevant headers are protected by OSCORE. Still, some other security-critical header entries including the Token Length, Message ID, Token and Max-Age option as well as the meta information for the body length are left unprotected. The reason why leaving out the first four meta data lies in the fact that these entries may be changed by intermediate nodes. However, not considering these meta data elements opens the door for man-in-the-middle attacks. Possible attack vectors can be spoofing the Message ID and Token in order to provoke a mismatch between requests and responses. Also, sending the Max-Age option without any protection is critical, as it has a similar functionality like the Cache-Control header in HTTP. When it gets tampered, the freshness of the response is corrupted analogously to attack #6 in Table 2.1. On the contrary, signing these header entries to avoid the aforementioned attacks prevents middleboxes from changing these elements. Thus, leaving out these header entries from the protection sphere and protecting these meta data elements induce issues on both sides. To resolve this problem, an enhanced approach has to ensure the integrity of these header entries and it must allow intermediate systems to modify the meta data elements simultaneously. Moreover, OSCORE does not consider protecting the integrity of the body length meta information. The reason behind this might be that no header for the body length is specified in the CoAP standard, as this information must be extracted from the UDP packet. This omission enables the manipulation of the body as manifested by attack #4 in Table 2.1.

According to the specification, messages protected by OSCORE are not intended for being cached. Each response is strictly bound to its corresponding request. Not supporting the option to cache messages may lead to a low scalability and violates the cacheability principle, which is one of the most vital REST constraints. Also, the OSCORE specification does not provide a protection for acknowledgment and reset messages. As both messages are utilized to confirm or reject a CoAP request or response, they must be secured as well. This prevents attacker from replacing an acknowledgment message by a reset message or vice versa. Another issue of OSCORE is a missing description that lists additional or application-specific header entries to be signed and/or encrypted.

#### **2.5.16 Granjal et al.**

Granjal et al. [GMS13] propose a scheme that signs and/or encrypts CoAP messages. This approach offers the options to encrypt a message, sign a message or sign as well as encrypt a message. However, the authors do not provide any policy, which specifies a list on the to be protected header entries. The proposed approach computes a signature and an encryption over the entire CoAP message including the payload and all present header entries. The resulting cipher-text and signature value are then stored in newly defined security headers which contain information on the security context such as the key type, whether the message is encrypted, signed or both and the destination. The latter information can refer to endpoints, i.e., client and server, or intermediate systems. A CoAP message may include one or multiple instances of these security header entries. Thus, the approach of Granjal et al allows to compute signatures and cipher text for multiple endpoints and intermediaries which enables an intermediate system to verify and decrypt traversing messages. However, the paper does not describe whether an intermediary is able sign and decrypt a message itself. This is an important property of a REST-ful security scheme in order to comply with the layered system constraint. This principle enables intermediate systems to interpret and transform the content of a message. The ability of intermediaries to sign and encrypt messages by themselves allows them to transform messages or part of it and inform the endpoint that a distinct intermediary has processed certain message elements. Moreover, encrypting and signing the entire message without obeying a policy, which defines what headers are protected, violates the self-descriptive constraint. This prevents certain intermediate nodes from accessing and modifying a message. That is, the signature of the message is invalid if an intermediary changes the message, as no policy for describing the modification exists. The other drawback is that a completely encrypted message is not accessible by intermediate systems not possessing the required decryption key. Both scenarios may occur, as a lot of intermediaries are either transparent or reside outside organizational boundaries of the client and the server.

#### **2.5.17 Consolidated Review of Analysis Results**

The obtained insights from the conducted analyzes of the available related work are summarized in Table 2.2 and Table 2.3 . Note, that only approaches are listed, which ensure authenticity and integrity of distinct message elements. The schemes proposed in [Res15; SAB15b], [FHT15], [Mel16], [Oiw+17a], [De +14], [Sto+09], [SH13], [Gra+11] and API-keys are omitted for readability reasons, since they do not provide any integrity protection for headers.



Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⊗ not specified, - not required

message elements to be signed	Amazon [Ama19b]								Google [Goo17]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	◐	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Content-Type	◐	-	◐	-	-	-	-	-	◐	-	◐	-	-	-	-	-
Content-Length	◐	-	◐	-	-	-	-	-	-	-	-	-	-	-	-	-
Transfer-Encoding	◐	-	◐	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	●	●	●	●	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○
message elements to be signed	Microsoft [Mic17]								HP [Hew14]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	-	-	-	-	-	-	-	-	○	○	○	○	-	-	-	-
Method	●	●	●	●	-	-	-	-	○	○	○	○	-	-	-	-
Status code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Content-Type	◐	-	◐	-	-	-	-	-	◐	-	◐	-	-	-	-	-
Content-Length	◐	-	◐	-	-	-	-	-	-	-	-	-	-	-	-	-
Transfer-Encoding	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	○	○	○	○	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○
message elements to be signed	OAuth v2 MAC Tokens [RMT14]								Signing an HTTP Request ... [RBT16]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	○	○	○	○	-	-	-	-	○	○	○	○	-	-	-	-
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status Code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	◐	-	-	-	-	-	-	-	◐	-	-	-	-	-	-
Content-Type	◐	-	◐	-	-	-	-	-	◐	-	◐	-	-	-	-	-
Content-Length	◐	-	◐	-	-	-	-	-	◐	-	◐	-	-	-	-	-
Transfer-Encoding	◐	-	◐	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	●	●	●	●	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○
message elements to be signed	Signing HTTP Messages [CS19]								OAuth v1 [Ham10]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	○	○	○	○	-	-	-	-	○	○	○	○	-	-	-	-
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	◐	-	-	-	-	-	-	-	◐	-	-	-	-	-	-
Content-Type	◐	-	◐	-	-	-	-	-	◐	-	◐	-	-	-	-	-
Content-Length	◐	-	◐	-	-	-	-	-	○	-	○	-	-	-	-	-
Transfer-Encoding	◐	-	◐	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	●	●	●	●	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	●	●	●	●	-	-	-	-	○	-	○	-	-	-	-	-
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○
message elements to be signed	Serre et al. [Ser+12]								HTTP Digest Authentication [SAB15b]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	○	○	○	○	-	-	-	-	○	○	○	○	-	-	-	-
Method	◐	◐	◐	◐	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	○	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-Type	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Content-Length	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Transfer-Encoding	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	○	○	○	○	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	●	●	●	●	-	-	-	-	○	-	○	-	-	-	-	-
Time variant parameter	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
message elements to be signed	Peng et al. [PLH09]								Lee et al. [LJK15; LJK17]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	○	○	○	○	-	-	-	-
Version number	○	○	○	○	-	-	-	-	○	○	○	○	-	-	-	-
Method	●	●	●	●	-	-	-	-	○	○	○	○	-	-	-	-
Status code	-	-	-	-	○	○	○	○	○	○	○	○	○	○	○	○
Connection	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Cache-Control	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Accept	-	-	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-Type	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Content-Length	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Transfer-Encoding	○	-	○	-	-	-	-	-	○	-	○	-	-	-	-	-
Host	○	○	○	○	-	-	-	-	○	-	○	-	-	-	-	-
Hash of body	○	○	○	○	-	-	-	-	○	-	○	-	-	-	-	-
Time variant parameter	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○

Table 2.2: Analysis of related work in HTTP message authenticity and integrity

Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⊗ not specified, - not required

CoAP message elements to be signed	OSCORE [Sel+18]								Granjal et al. [GMS13]							
	Request				Response				Request				Response			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
Version number	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Type	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Token Length	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Code (method code, response code)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Message ID	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Token	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Uri-Host, Uri-Port, Uri-Path	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Max-Age	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Location-Path, Location-Query	-	-	-	-	-	-	-	-	-	-	-	-	●	●	●	●
Accept	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Content-Format	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Body length (Payload-Length)	○	○	○	○	○	○	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Body	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Table 2.3: Analysis of related work in CoAP message authenticity and integrity

The related work analysis reveal that a lot of REST-based HTTP and Coap message authentication attempts have been evolved so far. However, none of the examined approaches targets the same abstraction layer as REST. Also, the evaluated mechanisms contain many vulnerabilities or are not in conformance with the REST constraints.

The concrete adoptions to Web, Cloud and IoT services are very diverse, emphasizing the need for a more methodical approach to REST message authentication and to REST-Security in general. Moreover, due to the lack of a general REST-Security framework, the same situation can be expected to take place in any other appearing implementation domain, in which REST gets adopted. All this emphasizes the need for a more advanced and elaborated security for REST-based service systems.

## 2.6 Towards a General REST-Security Framework

The previous sections motivated and highlighted the need for a general REST-Security framework. The available approaches provide security solutions for REST-ful HTTP and REST-ful CoAP only and do not offer any concepts residing on the same abstraction layer as REST itself. Moreover, the introduced and discussed specifics of REST-based services of any kind made apparent, that the application of the available standards, technologies and research is neither developed in a manner that suits REST nor evolved enough in maturity for an adoption in security-sensitive or mission-critical environments.

This section, therefore, proposes a methodology for defining general REST-Security framework components. It starts by developing a generic authentication scheme for REST messages. This security concept marks an initial step towards a REST message security, which forms the vital foundation for the general REST-Security framework. However, before being able to design any security schemes for REST, the specifics and constraints of the architectural style require to be addressed first.

The REST message elements as well as the resource identifier forming the uniform interface can be implemented by different standards and are equally important for the message processing. Hence, a REST-Security scheme needs to consider them all in order to avoid otherwise possible vulnerabilities (see Section 2.4.3). Such security specifications must be defined on the same abstraction layer as REST itself, so that they can be applied to any concrete protocol instantiation in a methodical manner (see Figure 2.8). To do so, a formal description of REST messages and an identification of security relevant parts in such messages need to be at hand [LN15a].

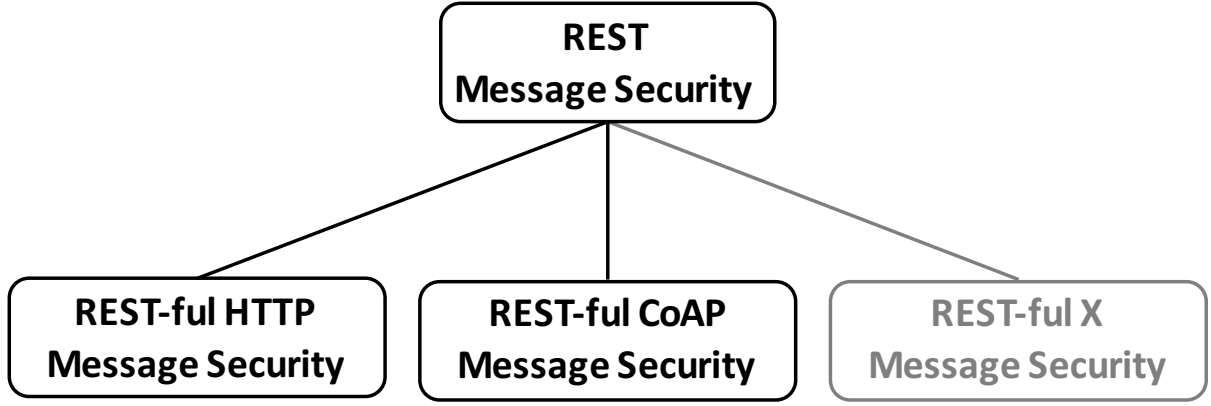


Figure 2.8: General REST message security and its instantiation to concrete REST-ful protocols

### 2.6.1 Formal Description of REST Messages

Since REST is constrained to the client-server model in conjunction with the request-response model, it is always the client issuing a request message to which the server replies with a corresponding response. The request message space is denoted by  $R_c$  and the response message space is referred to as  $R_s$  respectively. The whole REST message space  $R$  is henceforth

$$R := R_c \cup R_s. \quad (2.1)$$

The meta data space  $M$  is composed of the set of resource meta data  $M_r$ , the set of resource representation meta data  $M_b$  and the set of control data  $M_c$ :

$$M := M_r \cup M_b \cup M_c. \quad (2.2)$$

The control data set  $M_c$  consists of the set of request actions  $M_{ca}$ , the set of response meanings  $M_{cm}$ , the set of message parameterisation  $M_{cp}$  and the set of data to overwrite the default processing of a message  $M_{co}$ :

$$M_c := M_{ca} \cup M_{cm} \cup M_{cp} \cup M_{co}. \quad (2.3)$$

A REST message  $r \in R$  consists of two parts: a header  $h$  containing meta data and a body  $b$  comprising a resource representation. With  $H$  denoting the header and  $B$  the body space, the structure of a REST message is defined as

$$r := h || delimiter || b, \{(r, h, b) : r \in R \wedge h \in H \wedge (b \in B \vee b \in \emptyset)\}, \quad (2.4)$$

where *delimiter* is a set of characters separating the header from the body and  $||$  representing the concatenation operation. Note, that the actual embodiment of the delimiter depends on the concrete implementation of the uniform interface, i.e., the service protocol. In case of a binary protocol, the delimiter set might even be empty. For the sake of readability but without the loss of generality, the following explanations will focus on text-based protocols only, since these protocols include additional challenges in terms of the ordering, normalization and separation of

headers. To obtain an according description for binary protocols, these aspects can simply be omitted.

A header  $h$  holds a subset  $\dot{M} \subset M$  of the meta data entities:

$$h := \begin{cases} (\dot{M}, i), & \text{if } r \in R_c, \\ (\dot{M}), & \text{if } r \in R_s. \end{cases} \quad (2.5)$$

If  $h$  is part of a request message, it additionally includes a resource identifier  $i \in I$ , where  $I$  defines the set of resource identifiers. The constitution of  $h$  can further be concretized by the following policy:

- A message  $r \in R$  comprising a resource representation must include at least the two resource representation meta data entities  $m_{bl} \in M_b$  and  $m_{bt} \in M_b$  describing the length and the media type of the contained resource representation respectively.
- A request  $r \in R_c$  must contain at least one control data element  $m_{ca} \in M_{ca}$  and one resource identifier  $i$  describing the action and the target of the action respectively.
- A response  $r \in R_s$  must contain one control data element  $m_{cm} \in M_{cm}$  expressing the meaning of the response.

On the basis of this formal description, the following subsections introduce two generic schemes for ensuring the authenticity, integrity and non-repudiation of REST messages.

## 2.6.2 REST Message Authentication (REMA)

Following the introduced methodology and the results obtained from the related work analysis, the general REST Message Authentication (REMA) can be instantiated to REST-ful protocols of any kind (see Figure 2.9).

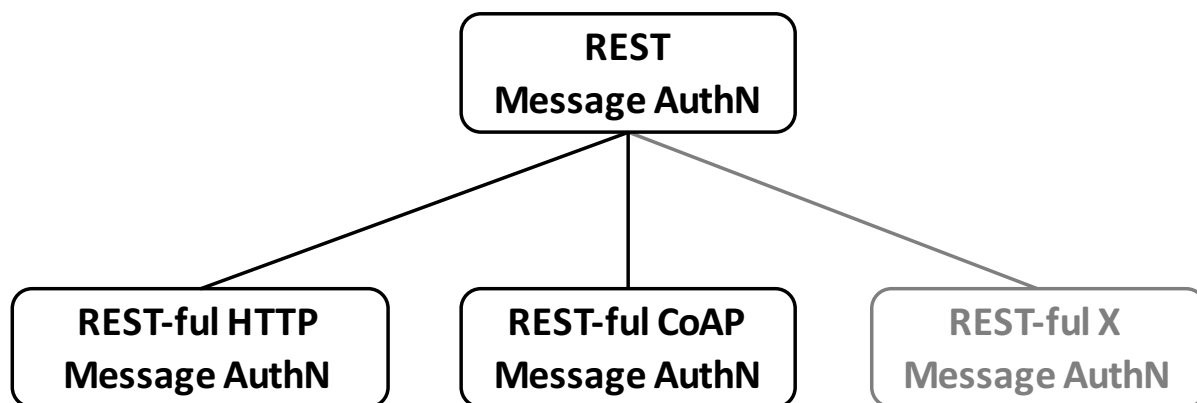


Figure 2.9: General REST message authentication and its instantiation to concrete REST-ful protocols

In order to illustrate the methodology, a REST-ful HTTP Message Authentication (REHMA, see Section 2.7.1) and a REST-ful CoAP Message Authentication (RECMA, see Section 2.7.2) are derived from the general framework subsequently.

## Message Parts to be Authenticated

The listed headers in the policy of Section 2.6.1 are crucial for the intended message processing and therefore need to be protected. In the following, the set of header entries containing the security-relevant and to be protected headers is denoted as  $\tilde{h}$ . Note, that  $\tilde{h}$  varies depending on whether it is part of a request or response, the action of the request, the meaning of the response and whether the message contains a resource representation or not. The variability of  $\tilde{h}$  can be especially substantiated by the request actions. Depending on the objective of the action,  $\tilde{h}$  requires a different set of meta data.

The following rules extend the policy of Section 2.6.1 and define additional security-relevant and mandatory headers to be authenticated and integrity protected for service protocols supporting CRUD actions. The combined rules are henceforth denoted as the REMA policy.

- A read request must contain at least one resource representation meta data element  $m_{br} \in M_b$  describing the desired media type being requested. Moreover, this request must not include a resource representation.
- A creation request must contain a resource representation.
- An update request must contain a complete or partial resource representation.
- A delete request does not require any additional prerequisite headers until further requirements. Moreover, this request must not include a resource representation.

Further extension of the REMA policy in terms of additional security-relevant header entries contained in  $\tilde{h}$  are a matter of the technical instantiation of REST and the application domain. Based on these abstract notations, a general signature generation and verification scheme for REST messages can be defined.

## REST Message Signature Generation

Algorithm 1 defines a general method for ensuring the authenticity and integrity of REST messages by generating a digital signature over the body and security-vital header entries as defined above. Note, that error conditions are not made explicit for readability reasons. Each error will cancel the signature generation process with an according error message.

As input, the signature generation algorithm requires a REST message  $r$ , a signature generation key  $k$  and a description  $desc$ . The latter parameter contains application-specific headers, which are to be appended to  $\tilde{h}$ . After obtaining the body  $b$  and the header  $h$  from the message  $r$ , the function in line 3 checks by means of the REMA policy that all required header entries are included in  $h$  and if so, constructs  $\tilde{h}$  out of them. Then eventually specified additional headers in  $desc$  are appended to  $\tilde{h}$ . In order to avoid replay attacks, the signature generation algorithm creates of a fresh time-variant parameter  $tmp$ . This parameter is the first element to be assigned to the  $tbs$  variable, which is gradually filled with the data to be signed. These two steps must not be omitted even when a concrete instantiation of this scheme already includes a time-variant parameter in  $\tilde{h}$ , since between message generation and signature generation might exist a considerable time spread. All headers contained in  $\tilde{h}$  are normalized and concatenated to  $tbs$ . In order to tie the resource representation  $b$  to  $\tilde{h}$  inducing the integrity of the conjunction of security-relevant header entries and body, it needs to be appended to  $tbs$  as well. The resource representation  $b$  is therefore hashed by a cryptographic hash function and the resulting hash

value is attached to  $tbs$ . Note, that in case a message does have an empty resource representation, a hash of an empty body is computed and added to  $tbs$ . The next statement signs the crafted  $tbs$  with a signature generation key  $k$ . Algorithm 1 outputs the generated signature value  $sv$  and the time-variant parameter  $tvp$ .

---

**Algorithm 1** REST Message Signature Generation [LN15a]

---

**Input:** REST message  $r$ , description  $desc$  of the application-specific header entries to be signed, signature generation key  $k$

**Output:** Signature value  $sv$ , time-variant parameter  $tvp$

```

1:  $b \leftarrow getBody(r)$ 
2:  $h \leftarrow getHeader(r)$ 
3:  $\tilde{h} \leftarrow getTbsHeaders(h)$ 
4:  $\tilde{h} \leftarrow \tilde{h} \parallel getTbsHeaders(h, desc)$ 
5:  $tvp \leftarrow generateTimeVariantParameter()$ 
6:  $tbs \leftarrow tvp$ 
7:  $i \leftarrow 0$ 
8: while  $i < |\tilde{h}|$  do
9:    $tbs \leftarrow tbs \parallel delimiter \parallel normalize(\tilde{h}_i)$ 
10:   $i \leftarrow i + 1$ 
11: end while
12:  $tbs \leftarrow tbs \parallel delimiter \parallel hash(b)$ 
13:  $sv \leftarrow sign(k, tbs)$ 

```

---

With these two outputs, an authentication control data element  $m_{cpa} \in M_{cp}$  can be generated, containing the signature algorithm name  $sig$ , the hash algorithm name  $hash$ , a key identifier  $kid$ , the time-variant parameter  $tvp$ , the signature value  $sv$  and the presence of additional header entries given by  $desc$  in the specified order. This control data element  $m_{cpa}$  needs ultimately to be embedded into the respective message  $r$ . Since resource representations can vary,  $m_{cpa}$  must be integrated into the header  $h$  of the message  $r$  in order to remain data format independent.

## REST Message Signature Verification

Algorithm 2 specifies the signature verification procedure for REST messages signed by Algorithm 1. The signature verification algorithm requires a signed REST message  $r$  as input and it returns a boolean value expressing the signature validation result. From the signed message  $r$  the required parts are extracted, including the message body  $b$  and the message header  $h$ . From  $h$  the authentication control data header  $m_{cpa}$  is obtained next containing the concatenated values  $sig$ ,  $hash$ ,  $kid$ ,  $tvps$ ,  $sv$  and  $desc$ . After building  $\tilde{h}$  in line 5, the next statement appends the additional header entries defined in  $desc$  to  $\tilde{h}$  in order of appearance. Then the headers in  $\tilde{h}$  are iterated in the same manner—and especially the same order—as during the signature generation process to build  $tbs$ . With  $tbs$  and the signature verification key identifier  $kid$ , the verification of the signature value  $sv$  can be performed. The boolean verification result is assigned to the variable  $valid$ , which represents the output of the signature verification procedure.

---

**Algorithm 2** REST Message Signature Verification [LN15a]

---

**Input:** Signed REST message  $r$ **Output:** Boolean signature verification result  $valid$ 

```
1:  $b \leftarrow getBody(r)$ 
2:  $h \leftarrow getHeader(r)$ 
3:  $m_{cpa} \leftarrow getAuthenticationControlData(h)$ 
4:  $(sig, hash, kid, tvp, sv, desc) \leftarrow split(m_{cpa})$ 
5:  $\tilde{h} \leftarrow getTbsHeaders(h)$ 
6:  $\tilde{h} \leftarrow \tilde{h} || getTbsHeaders(h, desc)$ 
7:  $tbs \leftarrow tvp$ 
8:  $i \leftarrow 0$ 
9: while  $i < |\tilde{h}|$  do
10:    $tbs \leftarrow tbs || delimiter || normalize(\tilde{h}_i)$ 
11:    $i \leftarrow i + 1$ 
12: end while
13:  $tbs \leftarrow tbs || delimiter || hash(b)$ 
14:  $verify \leftarrow getVerificationAlgorithm(sig)$ 
15:  $valid \leftarrow verify(kid, tbs, sv)$ 
```

---

### 2.6.3 REST Message Confidentiality (REMC)

In layered systems such as constrained by REST, confidentiality is of specific importance, since intermediate systems otherwise have plain-text access to traversing messages and those systems most commonly reside outside organizational boundaries of service providers and consumers. To prevent intermediaries from accessing sensitive message parts, the encryption of REST messages is a required foundational REST message security building block.

REMA ensures the authenticity, integrity and—when using asymmetric digital signatures in conjunction with a suitable PKI—non-repudiation of REST-ful protocol messages. In order to approach a comprehensive REST message security, the confidentiality must be taken into account as well. Following the introduced methodology of this paper, a REST message confidentiality scheme has to define a general policy and algorithms for protecting REST messages from unauthorized data disclosure. Such a scheme then serves as a guideline for adapting and implementing confidentiality services for concrete REST-ful technologies including HTTP, CoAP and prospective ones (see Figure 2.10).

In contrast to REMA, we do not specify the complete REMC framework, as this is not required to proof the proposed concept. REMA is sufficient and more suitable for this purpose, since there is much more related work available that can be used for evaluation. Still, we want to briefly discuss the requirements and challenges REST message confidentiality framework needs to tackle.

Encrypting the whole REST message—so that only the endpoints can read and interpret the intention of it—does not conform with the self-descriptive messages and layered systems constraint, though. As mentioned before, the both principles require that the semantics of REST messages have to be visible to intermediaries for enabling intermediate processing [Fie00]. This means, the key challenge of a REST message confidentiality scheme is to shield REST message from unauthorized data disclosure while retaining the self-descriptiveness for endpoints and



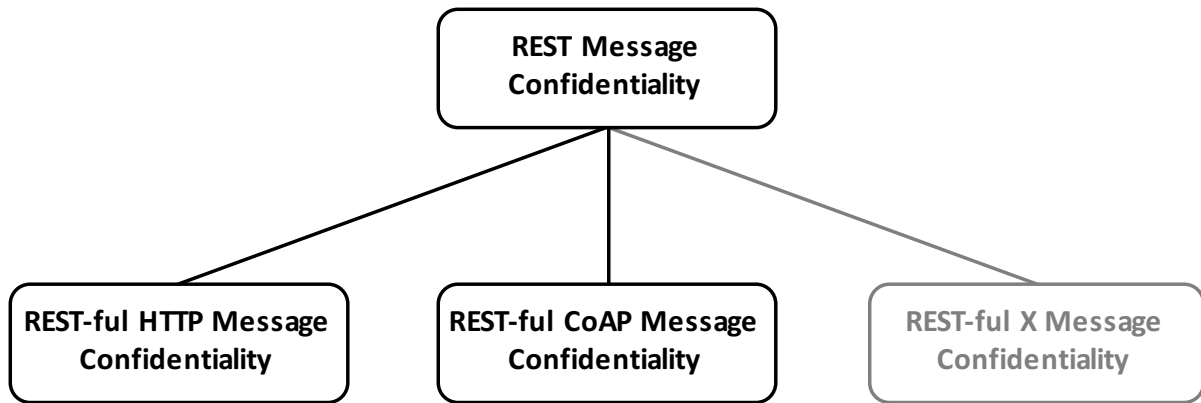


Figure 2.10: General REST message confidentiality and its instantiation to concrete REST-ful protocols

authorized intermediate systems. Hence, distinct message elements, which are required for a particular intermediary in order to render the message self-describing, must remain accessible for this respective intermediary. Consequently, a general policy for encrypting REST messages need to consider and specify what message parts are required to be accessible for which class of intermediaries.

This is especially true for caches, which must be able to read required message elements in order to store responses. As cacheability represents one of the core REST constraints for ensuring scalability, encrypted REST messages must therefore still provide the option to be cacheable. This aspect is, e.g., neglected by OSCORE [Sel+18] and [LJK15; LJK17]. The first approach does not consider cacheability of messages protected by OSCORE. The latter mechanism encrypts a REST message as a whole inducing so that an intermediate cache system is not able to interpret and store the message.

The body of a REST message is special in this context. In some cases it may contain a resource representation in others it does not. As XML, JSON and CBOR [BH13] are prevalent data formats for the resource representation in REST-based service systems, such a resource representation might already been encrypted by an according data encryption technology, such as XML Encryption [Ima+13], JSON Web Encryption [JH15] and CBOR Encryption [Bor17] respectively. Independent of an existing application-controlled resource representation encryption, the REST layer needs to incorporate own mechanisms for ensuring the confidentiality of the body. This is especially important for resource representations which do not include an encryption scheme such as HTML [Hic+14], YAML [BEN09] or CSV [Sha05]. As discussed before, the access to the body can then be granted to classes of intermediaries requiring it.

All these aspects will be elaborated in future work in order to develop a REST message confidentiality scheme that is in conformance with the architectural principles and constraints of REST. Combining such a REST-ful message confidentiality with the introduced REST-ful message authentication provides the fundamental layer of the REST-Security stack depicted in Figure 2.5.

## 2.7 Implementation of REST Message Authentication

To proof the proposed conceptual approach, we introduce two distinct instantiations of the general REST message authentication framework REMA, one for HTTP (see Section 2.7.1) and one for CoAP (see Section 2.7.2). Both concrete REST-ful message authentication schemes are intended to evaluate the coherent security construction in both distinct protocols and the robustness against the observed vulnerabilities contained in the current state of the art.

### 2.7.1 REST-ful HTTP Message Authentication (REHMA)

This section introduces the REST-ful HTTP instantiation of REMA denoted as REST-ful HTTP Message Authentication (REHMA). The following table emphasizes the instantiation of the generic REST message signature generation algorithm of [LN15a] as presented in Section 2.6.2 for HTTP requests and responses. This implementation uses string concatenation to build the string to be signed (*tbs*), which consists of a time-variant parameter (*ttp*), security-relevant header entries and the hash of the body. Note that for the specific instantiation to HTTP, the delimiter becomes the newline character '\n' and the binary hash value needs to be text-encoded by making use of a Base64URL transformation [Jos06].

<i>tbs</i> string template for HTTP request	<i>tbs</i> string template for HTTP response
<code>ttp + "\n" +                      UpperCase(Method) + "\n" +                      RequestTarget + "\n" +                      UpperCase(Version) + "\n" +                      LowerCase(Header0) + "\n" +                      ...                      LowerCase(HeaderN) + "\n" +                      Base64URL(hash(Body))</code>	<code>ttp + "\n" +                      UpperCase(Version) + "\n" +                      StatusCode + "\n" +                      LowerCase(Header0) + "\n" +                      LowerCase(Header1) + "\n" +                      ...                      LowerCase(HeaderN) + "\n" +                      Base64URL(hash(Body))</code>

Assume, that the following example request and response messages require to be authenticated.

Example HTTP request message	Example HTTP response message
<pre>GET /courses HTTP/1.1 Host: example.org Accept: application/json Connection: keep-alive Cache-Control: max-age=3600</pre>	<pre>HTTP/1.1 200 OK Content-Length: 19 Content-Type: application/json Server: Apache Connection: keep-alive Cache-Control: max-age=3600 Transfer-Encoding: gzip  {"REST":"Security"}</pre>

Based on the definitions, rules and policies specified in [LN15a] and the templates shown in the previous table, the *tbs* strings are constructed for the request message as shown in the left column of the following table respectively for the response message as shown on the right:

<i>tbs</i> string of example HTTP request message	<i>tbs</i> string of example HTTP response message
<pre>2014-11-21T15:26:43.483Z GET /courses HTTP/1.1 application/json max-age=3600 keep-alive example.org 47DEQpj8HBSa...km5NmpJWZG3hSuFU</pre>	<pre>2014-11-21T15:26:45.351Z HTTP/1.1 200 max-age=3600 keep-alive 19 application/json max-age=3600 gzip mIxp6LC6E2cl...zHQQBHU_PI9zWBG8</pre>

The elements of the HTTP start line of the request and response respectively are added to *tbs* according to their predefined positions. The security-relevant header values are concatenated in alphabetical order of the header names. Note, that the construction of *tbs* does not include the Server header, since this meta data is—from an authenticity viewpoint—not a crucial information for the message processing.

The next step encodes the constructed *tbs* to UTF8 and signs the string with a key *k*. Since header entries in HTTP must be text, a transformation of the binary signature value to a text-based equivalent is required. This implementation uses a URL-safe Base64 transformation.

$$sv = \text{Base64URL}(\text{sign}(k, \text{UTF8}(tbs)))$$

The final step integrates the resulting text-encoded signature value *sv* along with the corresponding signature meta data to the newly defined Signature header.

Authenticated example HTTP request	Authenticated example HTTP response
<pre>GET /courses HTTP/1.1 Host: example.org Accept: application/json Connection: keep-alive Cache-Control: max-age=3600 Signature: sig=RSA/SHA256, ↳hash=SHA256, ↳kid=https://myid.org/cert, ↳tvp=2014-11-21T15:26:43.483Z, ↳addHeaders=null, ↳sigValue=&lt;sv&gt;</pre>	<pre>HTTP/1.1 200 OK Content-Length: 19 Content-Type: application/json Server: Apache Connection: keep-alive Cache-Control: max-age=3600 Signature: sig=RSA/SHA256, ↳hash=SHA256, ↳kid=https://example.org/crt, ↳tvp=2014-11-21T15:26:45.351Z, ↳addHeaders=null, ↳sigValue=&lt;sv&gt;</pre>
	<pre>{ "REST": "Security" }</pre>

Since both messages do not consider additional as well as application-specific headers to be protected by the signature, the *addHeaders* parameter within the Signature header, contains the value *null*. If further headers to be signed are required, a list containing the header names separated by a semicolon must be included. REHMA protects all HTTP messages against the attack vectors of Table 2.1. A Java implementation of REHMA is available at: <https://das.th-koeln.de/developments/jrehma>.

## 2.7.2 REST-ful CoAP Message Authentication (RECMA)

This section introduces the REST-ful CoAP instantiation of REMA denoted as REST-ful CoAP Message Authentication (RECMA). The following table shows the adoption of the REST-ful message signature algorithm defined in Section 2.6.2 for CoAP. It contains two templates, each constructing a byte concatenation (symbolized by ||) of a sequence of all security-relevant message elements including a time-variant parameter (*tvp*). The resulting concatenation is transformed into a byte array that is the *tbs* variable for CoAP. The implementation considers signing request and response messages as well as acknowledgment (T=0x02) and reset (T=0x03) messages. The template contained in the left column describes the construction rules of *tbs* for requests as well as responses and the template contained in the right column defines the construction of *tbs* for reset and acknowledgment messages.

<i>tbs</i> constructing template for CoAP request and response	<i>tbs</i> constructing template for CoAP ACK and RST
<pre> tvp   Version   Type   TokenLength   Code   MessageID   Token   Options0 ...   OptionsN   hash (Body) </pre>	<pre> tvp   Version   Type   TokenLength   Code   MessageID </pre>

Assume, that the following two example messages require to be authenticated. The message on the right is a POST request represented by the code value  $0x02$  ( $C=0x02$ ), which requires a confirmation by the server, whether the message has successfully been received. Confirmable messages are denoted by the message type value  $0x00$  ( $T=0x00$ ). Moreover, the example request contains the protocol version number  $0x01$  ( $V=0x01$ ), the message identifier  $0x01$  ( $MID=0x01$ ) and the token value  $0x0A$  that has a length of one byte ( $TKL=0x01$ ). The left example message is an acknowledgment message for the POST request specified on the left. It confirms the reception ( $T=0x02$ ) of a request identified by  $MID=0x01$ . The delimiter to separate the header from the body in all CoAP messages is  $0xFF$ .

Example CoAP request message	Example CoAP ACK message
<pre> V=0x01, T=0x00, TKL=0x01, C=0x02, MID=0x01 Token: 0x0A Uri-Path: 0x6974656D73 # items Content-Format: 0x32 Payload-Length: 0xF 0xFF {"item": "pork"} </pre>	<pre> V=0x01, T=0x02, TKL=0x00, C=0x00, MID=0x01 0xFF </pre>

According to the previous table and the requirements defined by [NL15] and [NL16], the *tbs* for both messages are constructed as follows:

<i>tbs</i> of example CoAP request message	<i>tbs</i> of example CoAP ACK message
<pre> 0x14D14486B51 #tvp   0x01 #Version   0x00 #Type   0x01 #TokenLength   0x02 #Code   0x01 #Message-ID   0x0A #Token   0x00 #Uri-Host (3)   0x00 #Uri-Port (7)   hash (0x6974656D73) #Uri-Path (11)   0x32 #Content-Format (12)   0x00 #Max-Age (14)   0x00 #Uri-Query (15)   0x0F #Payload-Length (65001)   hash (UTF8 ({"item": "pork"})) #Body </pre>	<pre> 0x14D14486B57 #tvp   0x01 #Version   0x02 #Type   0x00 #TokenLength   0x00 #Code   0x01 #Message-ID </pre>

The concatenation order of the CoAP start header items and the token follows the order of the predefined positions stated in the CoAP specification. The CoAP options are added in numerical order of the option numbers. Once, the construction of *tbs* for the two messages is finished, it is signed with the signature key  $k$ .

$$sv = \text{sign}(k, tbs)$$

The last step incorporates the resulting signature value *sv* and the corresponding signature meta data to newly introduced CoAP options, which are Signature-Value, Signature-Algorithm, Hash-Algorithm, TVP and a Key-ID.

Signed CoAP request message	Signed CoAP ACK message
V=0x01,T=0x00,TKL=0x01,C=0x02,MID=0x01 Token: 0x0A Uri-Path: "items" Content-Format: 0x32 Payload-Length: 0x0F Signatur-Algorithm: 0x01 Hash-Algorithm: 0x01 TVP: 0x14D14486B51 Signatur-Value: <sv> Key-ID: <kid> 0xFF {"item":"pork"}	V=0x01,T=0x02,TKL=0x00,C=0x00,MID=0x01 <b>Signatur-Algorithm: 0x01</b> <b>Hash-Algorithm: 0x01</b> <b>TVP: 0x14D14486B57</b> <b>Signatur-Value: &lt;sv&gt;</b> <b>Key-ID: &lt;kid&gt;</b> 0xFF

Both messages use numbers to declare the signature and hash algorithm name. Here, the number 0x01 within the Signature-Algorithm option represents an HMAC-SHA256 signature and the same number defines a SHA256 hash for the Hash-Algorithm option. A description on the additional and application-specific header entries is not present in both messages, since an acknowledgment must not contain further options besides the options for the signature description and the request does not intend to include additional header entries.

As the CoAP standard does not define a meta data element for defining the length of the body, RECMA uses the Payload-Length option to declare the size of the body [LS14]. Even though this option is not a standardized header, i.e., it is only a proposed draft specification, it is still considered in the authentication process of RECMA to detect attacks that try to forge the body similar to attack #4 in Table 2.1. Moreover, RECMA utilizes this option to comply with the self-descriptive messages constraint that requires to be transport independent [Fie00]. RECMA foils all attack vectors presented by [NL15]. A Java implementation of RECMA is available at: <https://das.th-koeln.de/developments/jrecma>.

## 2.8 Evaluation and Discussion

The proposed REST message authentication scheme and the requirements defined for implementing REST message confidentiality are the first steps towards a general REST-Security framework. The REMA-Policy defines mandatory message elements for requests and responses, which need to be available in order to render a message self-descriptive. As these message entities are mandatory, they are also security-critical. Hence message elements defined in the REMA-Policy must be signed in order to be protected against malicious modifications. REHMA [LN15a] as well as RECMA [NL15] apply the REMA-Policy as a security-baseline for identifying corresponding mandatory and security-critical HTTP and CoAP message elements respectively. Moreover, the policy on the to be signed message elements in REHMA and RECMA are extended by protocol-specific mandatory header entries which are required for the self-descriptiveness of HTTP and CoAP messages.

The concrete adoption of the REST message authentication scheme in HTTP and CoAP are therefore not vulnerable to the attacks defined in Table 2.1 as well as the threats detected by the related work analysis, since all essential and security-critical message elements are signed (see Table 2.4). Note, that the to be signed message elements defined in REMA, REHMA

and RECMA protect against attacks which are generally-valid for all REST-based applications implemented with HTTP, CoAP other REST-based protocols. If another REST-based protocol or a distinct application domain utilize additional security-critical header entries, these elements must be added either to the policy of the concrete adoption or to the list of application-specific header entries included in *desc* in order to thwart protocol- or application-specific man-in-the-middle attacks.

Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⊙ not specified, - not required

message elements to be signed	REHMA [LN15a]							
	Request				Response			
	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-
Version number	●	●	●	●	●	●	●	●
Method	●	●	●	●	-	-	-	-
Status code	-	-	-	-	●	●	●	●
Connection	●	●	●	●	●	●	●	●
Cache-Control	●	●	●	●	●	●	●	●
Location	-	-	-	-	●	-	-	-
Accept	-	●	-	-	-	●	-	-
Content-Type	●	-	●	-	●	-	●	-
Content-Length	●	-	●	-	●	-	●	-
Transfer-Encoding	●	-	●	-	●	-	●	-
Host	●	-	●	-	-	-	-	-
Hash of body	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●

message elements to be signed	RECMA [NL15]							
	Request				Response			
	C	R	U	D	C	R	U	D
Version number	●	●	●	●	●	●	●	●
Type	●	●	●	●	●	●	●	●
Token Length	●	●	●	●	●	●	●	●
Code (method code, response code)	●	●	●	●	●	●	●	●
Message ID	●	●	●	●	●	●	●	●
Token	●	●	●	●	●	●	●	●
Uri-Host, Uri-Port, Uri-Path	●	●	●	●	-	-	-	-
Max-Age	●	●	●	●	●	●	●	●
Location-Path, Location-Query	-	-	-	-	-	-	-	-
Accept	●	●	●	●	-	-	-	-
Content-Format	●	●	●	●	●	●	●	●
Body length (Payload-Length)	●	●	●	●	●	●	●	●
Hash of body	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●

Table 2.4: To be signed message elements by REHMA and RECMA

Table 2.4 illustrates the message elements signed by REHMA and RECMA. As the REMA-Policy as well as the corresponding adoptions in REHMA and RECMA cover all the to be signed header in order to avoid the documented vulnerabilities (see Section 2.4, Table 2.2 and Table 2.3), it can also be utilized as an analytical framework for the evaluation and enhancement of related work in HTTP/CoAP signature schemes. For instance, REHMA may serve as a guideline for adding the missing to be signed message elements of the HTTP signature schemes required by the cloud storage services of Amazon [Ama19b], Microsoft [Mic17], Google [Goo17] and HP [Hew14]. The signature schemes of these cloud storage providers will benefit from the security specification of REHMA, as it will increase the level of security. This is especially

important as many companies use the cloud storage services of Amazon, Microsoft, Google and HP in production.

This paper shows that a general REST-Security scheme builds the basis for generally-valid policies and requirements. By means of this common foundation, REST-ful security technologies can be implemented based on the same security-baseline. This methodology has been conducted for the REST message authentication. The implementation of REMA in HTTP and CoAP shows an increase of the level of security, as the documented vulnerabilities can be avoided. Other or future REST-based protocols such as RACS [Uri19] can use the same methodology for defining security schemes. An adoption of REHMA in RACS is proposed in [NL16]. The reader is henceforth referred to this paper for further details.

## 2.9 Conclusion and Outlook

REST is an established approach for designing distributed applications and service systems that scale at large. This is especially true for the Web while other domains are following likewise. At the same time, the areas of adoption increase in criticality, making the need for appropriate security measures a necessity. The application of transport-oriented security is by far not sufficient and needs to be supplemented by adjacent message-oriented security mechanisms. In the latter respect, REST behaves very specific in comparison to existing approaches such as SOAP in the Web Services domain. This renders a straightforward adoption of available schemes and technologies from this domain infeasible. This is due to REST being an abstract architectural style on the one hand, that can be applied to many distinct technologies and environments. On the other hand, the particularities of REST demand for tailored approaches and schemes in order to not contradict with the REST constraints.

The introduced methodology marks an important step towards a structured and controlled procedure for developing appropriate security means for REST-based service systems and applications. The practical applicability of the introduced methodology has been proven by an adoption of it to authentication. The introduced generic REST message authentication scheme has then been instantiated to the REST-ful protocols HTTP and CoAP. A comparison with the current state of the art revealed that the available technologies are inhomogeneous and contain many vulnerabilities or do not comply with the REST constraints. This further emphasizes the need for a general and methodical approach towards REST-Security as has been proposed by this paper. Finally, an initial attempt towards REST message confidentiality is introduced discussing requirements and specifics to be considered while developing the complete picture of a general REST message security framework.

More research and development efforts in REST-ful message security are required in order to reach the necessary understanding of an adequate REST-Security defined at the proper abstraction layer while considering the specifics of REST. This is especially essential, since message security for REST-based service systems builds the foundation of many high-level security components (see Figure 2.5). Moreover, a stable and robust REST-Security cannot only set the scene for a mature service security stack, but it can also enhance available REST-based security technologies including OAuth [Har12] and OpenID Connect [Sak+14], which still suffer from many vulnerabilities [YM13; SB12].

Future work will focus on elaborating the REST-Security framework in the light of aspects such as performance and scalability. This includes the cacheability of protected REST-ful messages.



Moreover, concepts that enable intermediate systems transforming signed and encrypted REST messages will be studied as well. This is an important feature in a REST-ful architecture, since transforming the content of a message is an essential property of the layered system constraint.

# Chapter 3

## On the Security Expressiveness of REST-based API Definition Languages

### Summary of this publication

**Citation** H. V. Nguyen, J. Tolsdorf, and L. Lo Iacono. *On the Security Expressiveness of REST-Based API Definition Languages*. In: *International Conference on Trust and Privacy in Digital Business (TrustBus)*. 2017. URL: [https://doi.org/10.1007/978-3-319-64483-7\\_14](https://doi.org/10.1007/978-3-319-64483-7_14)

**Status of Paper** Published

**Type of Paper** Research Paper (Conference)

**Ranking** CORE: B, Microsoft Academics: C

**Aim** This paper provides a systematic analysis of fifteen service description languages with a specific focus on their ability to express security policies.

**Methodology** The evaluation of the service description languages is based on five criteria: (1) the ability to describe security schemes via native service description elements, (2) the set of security schemes which can be defined by default, (3) the ability to extend the default set of security schemes, (4) the approach for defining the semantics of not natively supported security schemes, (5) available work extending the service description language with additional security description elements.

**Contribution** The obtained results reveal substantial limitations in all analyzed specification languages. The detected shortcomings motivate the need for more profound research to establish service description languages as a reliable approach for describing REST-based security schemes.

**Co-authors' contribution** See Paper 2 in Section 1.1.1.

### 3.1 Introduction

The Service-Oriented Architecture (SOA) [Erl07] paradigm defines an architectural principle for implementing interconnected software systems via service orchestration or service choreography respectively. Contemporary business applications [LRS02] are greatly relying on this paradigm. It provides the foundation for a dynamic process management, in which service consumers and service providers are able to discover and bind themselves without knowing each other in advance. In this context, the service description –also known as the service contract– plays a central role when it comes to selecting and invoking services properly [LRS02]. Such service invocations commonly involve the exchange of sensitive information across organizational boundaries and multiple distinct enterprises. The protection of such services and the incorporated datasets is henceforth a necessity, rendering tailored security safeguards mandatory for SOA-based business systems.

Distributed systems following the SOA principles have been most commonly realized by SOAP-based web services [Gud+07]. Here, a service contract is defined by means of the Web Service Description Language (WSDL) [Chr+00]. To declare security, the standardization body OASIS maintains the WS-SecurityPolicy specification [Nad+12]. This security framework includes extensions for describing security requirements and policies in WSDL. As WSDL and the extensions in terms of protection means provided by WS-SecurityPolicy represent a machine-readable data format for describing protected SOAP services, the interface definition language is often used by developers for automatic code generation. This facilitates the proper invocation of services as well as implementation of security properties. On the other hand, it reduces the likelihood of developers for making programming mistakes.

Over the last years, web services have been deployed following the architectural style Representational State Transfer (REST) [Fie00]. One measure of how the architectural style influences contemporary service systems is an analysis of the platform ProgrammableWeb which has been conducted by the authors of this paper. This evaluation reveals that around 76% of 15,000 analyzed APIs are REST-based. In contrast to SOAP, the widespread usage of web services following the REST principles is, however, lacking on a standardized language for defining the service contract and security policies in particular. The missing technical foundation for describing REST-based web services in a machine-readable form, hinders the automatic discovery of services. Moreover, it increases the effort of implementing and testing the service invocations as automatic code generation is not supported. This induces a higher probability of producing insecure code as exemplified by Sun and Beznosov [SB12] in terms of the widely adopted authorization framework OAuth [Har12].

With the aim of establishing a REST-based counterpart to WSDL, several description languages for REST-based web services have been proposed. However, the languages' abilities to describe REST-based web services are very diverse. This paper analyzes the currently available description languages with the focus on the ability to declare required security policies and protection means. Section 3.2 lays the foundation for a basic understanding of the architectural style REST and thereby briefly recaps its key properties and constraints. Even though REST is still missing a standardized and mature security framework, a set of security mechanisms have become well-established and are presented in the Section 3.3. Based on this background, Section 3.4 evaluates the features and abilities of available service description languages for REST-based web services in respect to their security expressiveness. The findings are discussed in Section 3.5 and Section 3.6 concludes this paper with an outlook on future research challenges.

## 3.2 Representational State Transfer (REST)

REST [Fie00] is a guideline for designing distributed systems. Adhering to the architectural constraints recommended by REST results in applications that are easy to use, maintain and scale. To ensure scalability the communication in REST must be stateless, cacheable and layered. Simplicity is realized by a uniform interface. This constraint governs that components within a REST architecture must communicate through a set of predefined actions enriched by meta data. The communication in REST is resource-oriented, meaning that each request targets a resource. In the context of REST, a resource can be any kind of information which maps to a static or varying set of values. Depending on the client's preference, a resource can be delivered in different resource representation. Moreover, each request must be self-contained, i.e., messages must include all required data elements describing the intention of the message, so that its semantic is self-explaining for every component within a REST architecture. This includes a resource identifier and action describing the target and intention of the request as well as further information including the state and resource representation meta data. The returned response may contain further resource identifiers which are embedded in the meta data and/or resource representation. These resource identifiers (links) and their description serve as a service description for clients to perform further requests to other resources. This property is known as Hypermedia As The Engine of Application State (HATEOAS).

The architectural principles of REST are fairly abstract and can be adopted with any suitable set of technologies. One technology which conforms to the REST principles is HTTP, since it is stateless, cacheable and contains a uniform interface which includes standard actions (methods) and meta data to express, e.g., the state and the cacheability. Moreover, HTTP is designed to transfer and obtain different resource representations including HTML [Hic+14], XML [Bra+08] or JSON [Bra17]. HTTP represents the key technology of the web which is (arguably) the largest distributed system of the world [Fie00]. By this, REST has proven of being an architectural style for building distributed systems that scale at large. Due to this and the other given arguments, REST gets adopted meanwhile in many other domains than the initial web applications. Among them are many driving environments for business applications including the Cloud and IoT systems [SHB14; LN15b]. Consequently, security mechanisms for protecting REST-based application and services are becoming increasingly important.

## 3.3 Security Schemes for REST-based Web Services

Only few standardized security technologies do exist for REST-based web services. The HTTP Basic and Digest Authentication [Res15; SAB15b] are the first two security schemes which have been published for web applications. HTTP Basic Authentication represents a login process to restricted resources via username and password embedded in a specifically defined *Authorization* header field. HTTP Digest Authentication requires a username and password as well, but does not transfer the cleartext password in the message header. Instead it deploys a challenge response scheme in form of a random number and a hash. The OAuth framework [Ham10; Har12] is a standardized protocol for authorizing third party applications for accessing resources of end users. Two versions of OAuth have been proposed so far. OAuth provides a set of flows for retrieving tokens from an authorization server. Based on these tokens client are able to invoke information of end users from a resource server. OpenID Connect [Sak+14] extends OAuth by means of a standardized authentication. This specification enables the option for clients to

validate the identity of end-users. Many identity providers apply OpenID Connect as a technical baseline. Beside HTTP Basic/Digest Authentication, OAuth and OpenID Connect, API-Keys are commonly used for authenticate requests. Usually, these keys are random generated tokens which must be saved by the service consumer and service provider. In each authenticated request, this API-Key must be included in the URL or in an HTTP header field.

These security schemes, however, merely provide authentication and authorization. Confidentiality, integrity and non-repudiation have not been covered by current standards yet. To protect the confidentiality and integrity of the communication, many REST-based web services utilize Transport Layer Security (TLS) [Res18]. As TLS only ensures transport-oriented security which does not provide an end-to-end protection in layered systems, many approaches targeting additional safeguards on the application layer have been published in the recent years. These approaches propose, e.g., HTTP message signature schemes protecting for the whole HTTP message [LN15a]. This kind of authenticity and integrity protection is applied by the cloud storage services of Amazon [Ama19b], Google [Goo17], HP [Hew14] and Microsoft [Mic17] as a complementary shield to TLS.

To gain an overview on the usage of protection schemes of other service providers, 11,500 REST-based web APIs listed in the web API directory ProgrammableWeb have been analyzed. The analysis reveals that 5,248 of the 11,500 REST-based web APIs require one or multiple authentication scheme for accessing their service. Table 3.1 gives an overview on the usage of authentication schemes of REST-based web APIs requiring authentication.

Authentication Mechanisms	Total	Percentage of APIs using Authentication
API-Keys	2711	52%
Unspecified	844	16%
Token	819	16%
HTTP Basic Authentication [Res15]	741	14%
OAuth 2 [Har12]	606	12%
OAuth 1 [Ham10]	173	3%
App ID	163	3%
Other/Custom	132	3%
Shared Secret	99	2%

Only mechanisms with a usage  $\geq 1\%$  are considered. APIs may support multiple authentication schemes.

Table 3.1: AuthN schemes used by listed REST-based APIs in ProgrammableWeb

The directory distinguishes twelve authentication schemes from which service providers can select a subset to declare the protection scheme deployed for their API. Table 3.1 shows the most relevant ones which are sorted according their frequency of utilization. One observation is that the most frequently applied protection mechanisms are the ones which are not standardized. As no further security mechanism description apart from the name is specified by an API entry within the ProgrammableWeb directory, the actual security schemes declared as *Unspecified*, *Token*, *Other/Custom* and *Shared Secret* remain opaque for the user at the first glance. In most cases, users have to visit the web page of the API operated by the service provider in order to get further human-readable only information on the protection mechanism and details on implementing the client counterpart to it. Such security schemes can be any kind of safeguards

ranging from proprietary approaches to not yet standardized technologies such as the HTTP message signature.

This current situation shows that the description of diverse security policies in a machine-readable form can be a great benefit for assisting developers in building security mechanisms properly.

Thus, the following section evaluates available REST-based web services description languages with a special focus on their ability to express standard and custom security mechanisms.

### 3.4 Description Languages for REST-based Web Services

The previous section highlights the need for describing security policies in machine-readable manner in order to aid developers in implementing secure code. Moreover, the definition of security policies must be extensible as many service providers utilize custom or not standardized security schemes. This section therefore analyzes available service description language for REST-based Web Services according to the following criteria:

1. The ability to describe security schemes via native service description elements
2. The set of security schemes which can be defined by default
3. The ability to extend the default set of security schemes
4. The approach for defining the semantics of not natively supported security schemes
5. Available work extending the service description language with additional security description elements

#### 3.4.1 WSDL

The XML-based Web Service Description Language 2.0 (WSDL 2.0) [Chi+07] is a W3C Recommendation. In conjunction with the introduced HTTP adjunction [Lew+07] it has been the first approach providing a description language for REST-based web services. In contrast to its predecessor WSDL 1.1 [Chr+00], it offers a more general way of describing web services and it is not limited to SOAP anymore.

WSDL 2.0 considers the integration of security schemes. However only HTTP Basic and Digest Authentication are natively supported. Not supported security schemes can be integrated via the definition of new XML schema definitions. The drawback of WSDL 2.0 is the fact that it has not been widely accepted by developers for describing REST-based web services [Ver+14]. This might be the reason why no specification updates and XML schema definitions for REST-based security mechanisms have been presented so far.

### 3.4.2 WADL

Web Application Description Language (WADL) [Hea09] is another XML-based interface definition language for REST-based web services.

WADL does not consider a native support for security mechanisms. But the description of security schemes is extensible via XML schema definitions and distinct child nodes for defining meta information such as header field as well as URL parameters. As with WSDL 2.0, WADL suffers from the problem that it is not widely adopted in the REST community [Ver+14]. Hence, neither XML schema definitions nor well-defined specification on security scheme do exist for WADL to date.

### 3.4.3 RSDL

RESTful Service Description Language (RSDL) [Rob+13] is an additional XML-based technology for defining REST-based web APIs.

The support of authentication schemes is provided by using the authentication element, which, however, is not defined any further. The RSDL specification shows only one example on describing HTTP Basic or Digest Authentication via the authentication element. As RSDL utilizes XML and XML schema, standard as well as custom security schemes can be integrated by defining new XML schema definitions. Unfortunately, it seems that the RSDL specification is not maintained anymore. The last version of RSDL stems from a paper [Rob+13] in 2013. Since then, no further work on this approach has been published. Therefore, there is a lack of XML schema definitions and tools for standardized and custom security mechanisms.

### 3.4.4 RADL

Similar to RSDL, RESTful API Description Language (RADL) [RSZ16] defines a documentation technology for REST-based web services which is based on XML as well.

Authentication mechanisms can be defined by an authentication element likewise. As with RSDL, the RADL specification does not specify the description of standard security mechanisms in the current version. Since RADL applies XML and supports XML schema, the missing security mechanisms can be included via XML schema definitions. As the current state of RADL is still a draft, a set of aspects are not completely defined. This is especially true for security schemes. Here, no XML schema definitions and examples about the definition on available security technologies are specified so far.

### 3.4.5 REST Chart

Another XML-based description language is REST Chart [LC11]. The aim of REST-Chart is to specify a REST-based web API over transitions which contain two input and one or multiple output elements. First input element defines the link to be invoked and the second one specifies required HTTP methods, meta data and an optional resource representation. The output elements describe resulting status codes of responses in conjunction with an optional embedded resource representation.



REST Chart does not provide a native support for security schemes. However, protection mechanisms can be specified via the aforementioned transitions. In case of a login process, the first input element contains link which refers to authentication endpoint. The other input element includes a control child element which specifies the HTTP method to start the transition. If the POST method is used, the input element may include a representation element which defines required media type and the schema of the credentials. The output indicates possible returning status codes and resource representations of responses. As REST chart does not specify an option for defining required header fields, this is the only way of describing a security process in the current state of REST Chart. The input element may include meta data nodes, but beside the fact that the meta data element can contain any kind of text-based XML attributes, the meta data element is not defined any further. Hence, the ability to describe an authentication scheme, which consider header fields for expressing the credentials, is therefore limited. Since REST Chart utilizes XML and therefore supports XML schema, this missing functionality can be included by new XML schema definitions. However, no specification and tools for such an extension have been proposed so far.

### 3.4.6 OAS / Swagger

The OpenAPI Specification (OAS) [Ope16], formerly known as Swagger [Sma16], represents a REST-based web service description languages which is not based on XML. The approach utilizes YAML [BEN09] or JSON [Bra17] as the technical foundation.

OAS provides a native description of security schemes. Security mechanisms are defined by the Security Definition Object which consists of multiple Security Scheme Objects. OAS supports HTTP Basic Authentication, API-Keys and OAuth 2.0 natively. Extensibility of not yet supported security mechanisms is, however, limited. Natively supported security schemes can only be extended by additional Security Scheme Object attributes. The integration of Security Scheme Objects defining new security schemes is not considered yet. Also, no work is available so far, which defines a definition approach for not supported security mechanisms.

The main strength of OAS is the wide range of tool support. Many technologies do exist for automatic testing and code generation which makes OAS well-established by developers.

### 3.4.7 RAML

RESTful API Modeling Language (RAML) [RAM16] is another YAML-based description language for REST-based web services.

As with OAS, the RAML specification considers the integration of security mechanisms. This is realized by the securitySchemes element which comprise one or multiple security schemes. Each security scheme must contain a type attribute which is the identifier of the mechanism. RAML natively supports the types OAuth 1.0, OAuth 2.0, Basic Authentication, Digest Authentication and Pass Through. API-Keys are not supported by default. Custom or not defined security schemes can be described via the *x-`<other>`* type, where *<other>* represents the placeholder for the security mechanism name. In the current version, RAML does not provide the option for appending a semantic description of security schemes with *x-`<other>`* types. This shortcoming restricts the definition of custom and not specified safeguards which might be the reason why no work on describing other security schemes have been published so far.

Similar to OAS, RAML promotes a lot of tools for, e.g., testing and automatic code generation. Therefore, this approach is also widely used by developers.

### 3.4.8 API Blueprint

API Blueprint [[Blu16](#)] is another widely used description language for REST-based web services alongside OAS and RAML. The syntax of this approach is based on MSON [[Api16](#)], which itself is based on Markdown [[Leo16](#)].

The authors of API Blueprint attempt to establish the description language as an RFC. Here, an authentication framework in draft status is proposed. This draft depicts the general definition on authentication schemes as well as a description on concrete mechanisms including HTTP Basic Authentication and OAuth. Using this authentication framework, other authentication schemes can be included to an API Blueprint service description. Furthermore, API Blueprint's approach to utilize MSON for its description is different to previous concepts. As Markdown represents definition syntax for producing human-readable content such as HTML or readme documents, and MSON introduces conversion of Markdown to JSON or XML documents, API Blueprint may also address machine-readability of security extensions to some extent. However, it does not cover the complexity of describing the semantics of new defined elements such as provided by JSON or XML schema definitions.

As with OAS and RAML, developers using API Blueprint benefits from a set of tools which aid them in testing and implementing REST-based web services.

### 3.4.9 OData

OData [[Han+16b](#)] is an OASIS standard for describing REST-based web services. OData services are defined via an Entity Data Model (EDM). This model contains vocabularies for specifying the data model of the resource representation and their relationships. Additionally, the EDM includes elements for describing actions and URL queries and paths. A service description in OData can be defined either in JSON [[Han+16a](#)] or in ATOM [[Har+13](#)]. The specification of the standard vocabulary is however defined in XML Schema. Similar to RAML, OAS and API Blueprint, OData provide a lot of libraries, SDKs and tools for implementing and testing services. Moreover, many service providers, among them also services of, e.g., Microsoft, IBM and SAP offer their service description via OData.

The OData specification recommends to use HTTP Basic authentication over TLS for securing REST-based OData services. Apart from the aforementioned authentication scheme no other security mechanisms are recommended or provided. To complement OData service descriptions with additional security policies, new XML Schema definitions can be used to extend OData. At the moment, no further security specification or work on integrating security in OData description do exist so far.

### 3.4.10 I/O Docs

I/O Docs [[TIB15](#)] is a JSON-based approach for documenting REST-based web services. Currently, the specification of I/O Docs is only based on examples. A general definition on

the I/O Docs elements is not defined yet. Also, no description on the definition of new service description elements does exist so far. This is especially true for defining security schemes. Only examples are available which show the definition on authentication schemes. Examples exist for HTTP Basic Authentication and OAuth, but other security mechanisms are not described. As with many aforementioned description languages, I/O Docs suffers from the low frequency of usage. This might be the reason why no extensions, tools and updates have been proposed recently.

#### 3.4.11 hRESTS and RDFa

HTML for RESTful Services (hRESTS) [KGV08] offers another approach as the aforementioned description languages. Instead of defining a new data format for describing REST-based web services, hREST augments HTML by including new HTML elements. The aim of this approach is to enrich HTML with machine-readable HTML elements, without modifying the visualization of the web page. These HTML elements provide additional information which can be processed by a machine-driven process. This has the advantage that a returned HTML contains human- and machine-readable description simultaneously. The semantics of new HTML elements for hRESTS is extensible via ontologies.

Resource Description Framework (RDF) is a model for describing machine-readable linked data structures and web APIs. The semantics of RDF elements and their relationships are defined by ontologies likewise. An RDF model can be implemented in various data formats such as XML. Resource Description Framework in Attributes (RDFa) [ABM15] defines an adoption of RDF in HTML attributes. As with hRESTS, the goal of RDFa is to enhance the machine-readability of HTML.

Neither hRESTS nor RDFa provide a native support for security mechanisms. However, protection means can be incorporated via ontologies. Maleshkova et al. [Mal+10] propose an approach on defining a new ontology for authentication schemes. The authentication ontology of [Mal+10] comprises limitation, tough, as it is composed of three classes only. These classes define the authentication mechanisms name, the credentials form (e.g. API-Key, username and password or OAuth token) and the transmission medium which specifies whether the credentials are include in the header or in the URL. Following this concept, the definition more complex security mechanisms such as the HTTP message signatures can not be implemented in straightforward manner, as no ontology element for describing a signature other cryptographic mechanisms is specified. To do so, the authentication ontology of [Mal+10] must be redesigned by including security services and additional security definition elements. Beside this publication, no other security-related work has been presented so far.

#### 3.4.12 ReLL

Resource Linking Language (ReLL) [AW10] is a REST-based web services description language that extends the vocabulary of RDF. ReLL utilizes XML to represent the RDF model.

ReLL does not consider a built-in support for security schemes. As it uses RDF, new elements and vocabularies can be added via ontologies in order to specify security schemes. Such an approach is presented by Bellido and Alarcon [BAS12]. Here, the authors introduce an example description on defining OAuth in ReLL. Both authors continue the work on defining security

schemes in [SAB15a] in which they propose ReLL-S, an ontology for describing security constraints and schemes. This ontology is more comprehensive than the ontology of [Mal+10]. It consists of a set of security goals which includes confidentiality, integrity, authentication and authorization. The security goals contain subclasses which defines cryptographic mechanisms (e.g. encryption and digital signatures) and authentication as well as authorization protocols. These subclasses include further subclasses referring to concrete security schemes such as OAuth, HTTP Basic/Digest Authentication as well as cryptographic algorithms such as AES, RSA or SHA. With the elements of ReLL-S, [SAB15a] introduces the definition of API-Keys, a simple username and password authentication, HTTP Basic/Digest Authentication, OpenID [RR06] as well as OAuth. As this ontology contains a comprehensive set of security elements, further security mechanisms can be deduced and included to a ReLL service description.

### 3.4.13 SERIN

Semantic RESTful Interface (SERIN) [de +13] is another description language for REST-based web services which is based on RDF. As with ReLL, SERIN also applies XML as the data format.

SERIN does not support any vocabulary for describing security policies by default. As SERIN is based on RDF, protection elements can be extended by introducing a new ontology. However, such extensions have not been proposed yet.

### 3.4.14 Hydra

Hypermedia-Driven API (Hydra) [Lan13; M L17] represents a W3C community group which attempts to establish a vocabulary for defining the semantic of linked data and web APIs. This approach is based on JSON for linked data (JSON-LD) [M S14], a specification for defining machine-readable semantics of data and links included in JSON. The vocabulary of JSON-LD elements is defined by the Schema.org community. Hydra extends the vocabulary of JSON-LD by defining elements and a schema for describing REST-based web services properties such as entry points, supported HTTP methods, URL query parameters and the meaning of status codes. The current version of Hydra does not consider the integration security mechanisms yet. As the specification of Hydra is an early stage, the description of security properties may be considered in future work. Currently, no external work which approaches to resolve this shortcoming does exist so far. For the time being, the semantics of security mechanisms can be described by defining new JSON-LD vocabularies.

### 3.4.15 RESTdesc

RESTdesc [Ver+12] is an academic approach that utilizes Notation3 (N3) [T B11] for describing REST-based web APIs. N3 is an extension of RDF.

The current status of RESTdesc does not consider the description of security mechanisms. As with RDFa and hRESTS, missing security schemes can be extended by the integration of new RDF ontologies. However, such extensions have not been published so far.

### 3.5 Discussion

The analysis of the previous section highlights that a lot of approaches to describe a service contract for REST-based web services have evolved over time. This already emphasizes the huge demand of such technologies. Still to date, there is no standardized language available for developers. This situation allows, nonetheless, to analyze the current proposals in order to conclude whether there are already mature and comprehensive technologies available which could serve as a basis for a standard or if there still exist research challenges that need to be solved first.

Description Language	Native support for security mechanisms	Natively supported security schemes	Extension approach	Available security extensions
WSDL 2.0 [Chi+07; Lew+07]	AuthN	HTTP Basic & Digest	XML Schema	-
WADL [Hea09]	-	-	XML Schema	-
RSDL [Rob+13]	AuthN	-	XML Schema	-
RADL [RSZ16]	AuthN	-	XML Schema	-
REST-Chart [LC11]	-	-	XML Schema	-
OAS [Ope16; Sma16]	AuthN, AuthZ	HTTP Basic, OAuth 2, OpenId Connect	-	-
RAML [RAM16]	AuthN, AuthZ	HTTP Basic & Digest, OAuth 1 & 2, Pass Through	x-<other> type	-
OData [Han+16b]	-	-	XML Schema	-
I/O Docs [TIB15]	AuthN, AuthZ	API-Key, OAuth 1 & 2	-	-
API Blueprint [Blu16]	AuthN, AuthZ	HTTP Basic, OAuth 2	-	-
hRESTS [KGV08] & RDFa [ABM15]	-	-	RDF Ontology	[Mal+10]
ReLL [AW10]	-	-	RDF Ontology	[BAS12; SAB15a]
SERIN [de +13]	-	-	RDF Ontology	-
Hydra [ML17]	-	-	JSON-LD Schema	-
RESTdesc [Ver+12]	-	-	RDF Ontology	-

Table 3.2: Security Expressiveness of REST-based API Definition Languages

Table 3.2 summarizes the available service description languages according to the five criteria defined in the previous section. Seven of the fifteen analyzed approaches consider the integration of security mechanisms. But only five evaluated technologies provide a native support for a set of standard security mechanisms. Most approaches offer the option to integrate missing security schemes by including or extending new schema and ontologies. Unfortunately, such security extensions are only available for hRESTS, RDFa and ReLL so far [Mal+10; BAS12; SAB15a]. The other technologies lack on further specification and work which extends these technologies by missing or additional security schemes. RAML and OAS support the description of many standardized security mechanisms. Driven by the global players of web technologies, both languages also provide a set of diverse tools for automatic code generation, testing and building REST-based applications and APIs. However, RAML and OAS do not provide the option for describing the semantics of other security schemes. This is also true for I/O Docs and API

Blueprint. The former technology does not define a specification aspect for describing extensions. API Blueprint lacks on the functionality for defining the semantics of new service description elements due to the usage of Markdown. WSDL 2.0, WADL, RDSL, RADL, REST-Chart, OData, SERIN and RESTdesc provide only a definition for few authentication schemes or no security mechanisms. Missing security definition can, however, be included by XML schema definitions or ontologies. Such extensions have not been present so far, though.

Moreover, almost all API definition languages only consider authentication and authorization. Description elements for defining confidentiality, non-repudiation, integrity and further security mechanisms are not supported by default. Only ReLL-S, the extension of ReLL, supports all aforementioned security services except non-repudiation. However, for ReLL and ReLL-S, no tools which support developers in implementing REST-based web applications do exist so far.

Also, no analyzed technology provides a description on the invocation properties of TLS. All approaches are only able to describe whether TLS is used or not. Properties such as supported cipher suites or the TLS version is not specified by any approach at all. As it has been shown that the implementation of transport-oriented security can cause many critical vulnerabilities [Fah+12; Geo+12] due to the high complexity, such a description could serve as basis for automatic code generation and testing. This set of tooling could assist developers in implementing proper TLS connections and may reduce the likelihood to make programming mistake.

Another missing aspect of all evaluated service description languages is the absent ability for describing security policies for the resource representation. A demand for such a security description is, for instance, needed in some OAuth and OpenID Connect environments, where JavaScript Object Signing and Encryption (JOSE) [IET17] is utilized for securing the tokens and other sensitive information. All service description languages supporting the description of resource representations only provide the declaration of the media type. These approaches can merely define that a resource representation embodies the media type application/jose+json, but the context and the semantics of the security policies can not be specified by any service description technology.

An additional shortcoming of all analyzed service descriptions is that they only provide a definition on REST-based services which use HTTP as the transfer protocol. None of them considers the description of services that utilize CoAP [SHB14], RACS [Uri19] or other REST-based protocols.

### 3.6 Conclusion and Outlook

Overall, the security expressiveness of the available REST-based web service description languages is still at its beginning. Besides authentication and authorization, there are no further security capabilities expressible by default and even these very basic protections are not provided by all of the analyzed languages (see Table 3.2). ReLL in conjunction with ReLL-S is the only approach which consider the integration of all standardized authentication and authorization schemes. Also, the ontology of ReLL-S provides service description elements for all security mechanisms except non-repudiation. The other evaluated service description technologies lack on a native definition of standard protection means or have restrictions in terms of extending and defining security mechanisms. Moreover, none of the evaluated approach provide a comprehensive description on TLS and the protection of the resource representation.



One reason for this current situation may lie in a lacking overall REST-Security framework [Gor+14a]. As current research activities are enhancing this field [LN15a; NL15; NL16], new REST-Security components may be evolved in the future. Hence, REST-based service description languages need to cope with this by an increased extensibility in respect to security-related expressiveness.

This shows that a bunch of research and development challenges still exist in order to find a service description language and a security policy framework for REST-based systems which can serve as a standard such as WSDL and WS-SecurityPolicy for the SOAP domain. As many service definition technologies have been proposed, further work will therefore focus on enhancing available languages in terms of security expressiveness and extensibility, instead of proposing a new approach. Also, future studies will analyze REST-based service description languages for other REST-based protocols including CoAP and RACS.



# Chapter 4

## Systematic Analysis of Web Browser Caches

### Summary of this publication

**Citation** H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Systematic Analysis of Web Browser Caches*. In: *2nd International conference on Web Studies (WS)*. 2018. URL: <https://doi.org/10.1145/3240431.3240443>

**Status of Paper** Published

**Type of Paper** Research Paper (Conference)

**Ranking** No ranking available

**Aim** In this paper, we introduce a tool-based approach for testing web caches. In particular, our approach includes a cache testing tool, a test suite containing 415 test cases as well as a test specification language which allows changing and extending test cases. We use this testing approach to analyze the compliance of the web browser caches in Chrome, Firefox, Safari and, Microsoft Edge.

**Methodology** To define a comprehensive test suite that covers all facets of web caching, we review the HTTP caching RFC 7234 and available literature on web caching.

**Contribution** The analysis shows many diversities as well as discrepancies. As our baseline of test cases is specified using a specification language that enables extensibility, developers as well as administrators and researchers can systematically add and empirically explore caching properties of interest even in non-browser scenarios.

**Co-authors' contribution** See Paper 3 in Section 1.1.1.

## 4.1 Introduction

The web can be considered as the world’s largest distributed system. Its ability to scale at large has been its formula of success ever since. To obtain high scalability, web caching systems are applied—among others—for optimizing network performance. A web cache is a subsystem for coordinating the transparent storage and retrieval of recyclable HTTP responses. By this, a web caching system potentially reduces three quantities: the number of requests that reach the origin server, the amount of network traffic resulting from document requests, and the latency that an end-user experiences in retrieving a document [Wan99]. Moreover, when serving recurring requests on behalf of an origin server that is not responding for some reasons, caches contribute to an increased availability of web-based services. Web caching systems can occur in various locations between the path from client to server. They can be implemented either as an external middlebox between the client and server application or as middleware which is included as an internal component in the client and server application. For instance, the web browsers Chrome, Firefox, Safari or Edge include a client-internal cache which store and reuses frequently requested web resources such as HTML documents, Javascript files, images or stylesheets.

For web developers and vendors of web browsers, the knowledge of and compliance with caching standards is crucial in many respects. Disobeying the standardized requirements and control directives impairs scalability and performance. Also, ignoring caching policies may induce security and privacy issues, if sensitive information are cached and reused although being prohibited. To prevent issues stemming from inappropriate web caching, one need to have a deep understanding of the current state of play. In Section 4.2 we therefore briefly recap key aspects of web caching. Based on these foundations, in Section 4.3 we give insights on web browser caches and the consequences of caching misbehavior. With the aim to detect compliance issues in web browser caches, we mandate for proper test tools that are currently lacking, as the related work review in Section 4.4 manifests. As a first contribution, we introduce a methodology for deriving meaningful test cases for auditing cache systems in Section 4.5. Following the proposed methodology, we were able to define 397 tests, which we compile to a general cache testing suite (see Section 4.6). With the purpose to evaluate the proposed approach and obtain a systematic analysis of available web browser caching systems, we conduct an empirical study of client-internal web caches residing in Chrome, Firefox, Safari and Edge. The main results are discussed in Section 4.7. Overall, they do affirm the relevance of appropriate cache testing tools supporting developers of caching components as well as developers instrumenting them in their web applications. We conclude this paper in Section 7.8 with an outlook on challenges in web caching.

## 4.2 Web Caching Foundations

The world’s largest distributed software system builds on HTTP [FR14c] as its fundamental communication technology for connecting web clients and servers. Between these two endpoints, various transparent intermediate systems such as caches may exist, performing specific tasks with the goal to optimize certain properties of the application.

RFC 7234 [FNR14] specifies web caching requirements, concepts and control directives in order to improve performance and resource utilization. Although caching in the web is optional, it can be assumed that reusing a cached HTTP response is the default behavior when no other policy

prevents it. Rather than mandating to store and reuse particular HTTP responses, the semantics of the headers specified in RFC 7234 are focused on preventing a cache from either storing a non-reusable response or reusing a stored response inappropriately. To lay the ground for the subsequent discussions, we summarize the core concepts and directives of RFC 7234 in the following.

#### 4.2.1 Explicit Caching

Caching systems entail the inherent risk that clients are provided with stale resources, i.e., resources which are out of date with respect to their master copies on the origin server. To hinder caches from reusing stale responses, the web caching standard defines a set of server-side and client-side control directives as well as headers for declaring explicit caching requirements.

RFC 7234 provides two conceptual approaches for ensuring the freshness of a cached resource. One is the server explicitly specifies the *freshness lifetime* of a resource, i.e. the time span between the generation of a response and its expiration time. An expired response is considered as stale. The other approach requires the caching system to check back with the origin server whether a cached resource is still fresh. This second approach is known as *freshness validation* or *conditional request*.

#### Explicit Freshness Lifetime

The conceptual message flow when applying the freshness lifetime is shown in Figure 4.1. When a request issued to a particular server traverses a caching intermediary, the cache checks whether it contains a fresh copy of the the requested resource. If it does not, it forwards the request to the server. The response from the server defines a freshness lifetime advising the cache to handle subsequent requests to this resource by its own for the specified amount of time. Thus, recurring requests are served by the caching system for the specified amount of time without requiring any further intervention with the origin server.

For explicitly declaring a freshness lifetime, RFC 7234 provides the `Expires` or `Cache-Control` headers. The `Expires` header specifies the absolute expiration time in form of a date whose format is defined in [FR14c]. The `Cache-Control` header field provides the `max-age` directive by which the relative freshness lifetime can be specified in seconds. For instance, `Cache-Control: max-age=60` within a response header defines that the corresponding response is fresh for 60 seconds.

Both declarations are recognized by all types of caches including shared and private ones. A *shared cache* stores and recycles responses for multiple users, while a *private cache* must only return stored responses to one single user. Content distribution networks (CDNs), forward and reverse proxies are typically shared caches. A prominent representative of the private cache family is the web browser cache. The `s-maxage` directive has the same functionality as `max-age`, with difference that it is intended for shared caches only. If the `Cache-Control` header field comprises `max-age` and `s-maxage` simultaneously, a shared cache has to consider the `s-maxage` directives while a private cache has to honor the value of `max-age`.

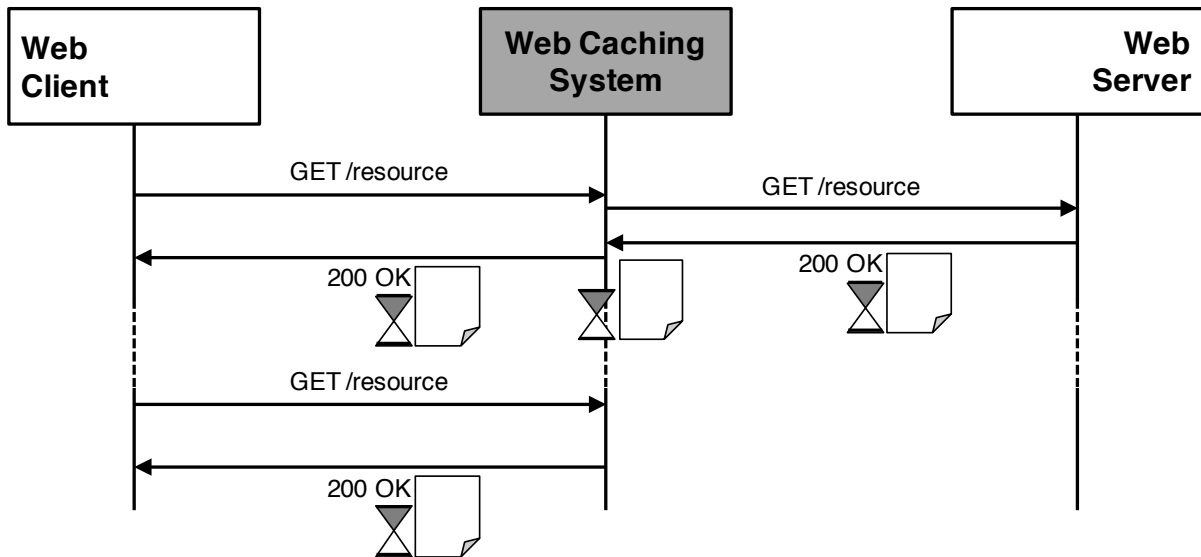


Figure 4.1: Functional principle of web caching with freshness lifetime. The response from the origin server contains meta data declaring the amount of time the response is allowed to be replayed by caches without any further server intervention.

### Explicit Freshness Validation

Beside the freshness lifetime, a cache can also infer the freshness of a response by a validation process (see Figure 4.2). A request for triggering a *freshness validation* is also known as *conditional request* [FR14a]. When a request issued to a particular web server traverses a web caching intermediary, the cache checks whether it contains a copy of the the demanded resource. If it does not, it forwards the request to the origin server in order to obtain the resource. The response from the server contains some kind of validation token identifying the version of the resource. For each subsequent request to this cached response the cache asks the origin server whether the resource is still up-to-date by adding the validation token to the request. In case the cached response is still valid, the server responds with the status code 304 Not Modified. Note, that the response body is empty as the cache is already in possession of a valid copy. If the validation request is unsuccessful, meaning that the requested resource has changed in the meantime, the origin server sends a response with the status code 200 Ok and a body holding the updated resource representation.

Origin servers can force caches to validate the freshness by including the `Cache-Control` header with the value `no-cache` or `must-revalidate`. The difference between those two directives lies in the fact that `no-cache` dictates a cache to validate the freshness of each request, while `must-revalidate` requires a verification only if the response exceeds the freshness lifetime. The `must-revalidate` directives also prohibits a cache from returning stale responses.

RFC 7234 define two types of validation tokens: (1) an opaque entity tag represented by the `ETag` header field and (2) a time-variant parameter that can be retrieved form the `Date` or `Last-Modified` header. If a response does not comprise the `ETag`, `Last-Modified` or `Date` header, a conditional request may contain a self-defined timestamp derived from a cache-internal clock. A freshness validation request intending to verify with an opaque token must contain the `If-None-Match` header comprising one or multiple entity tags. Freshness valida-

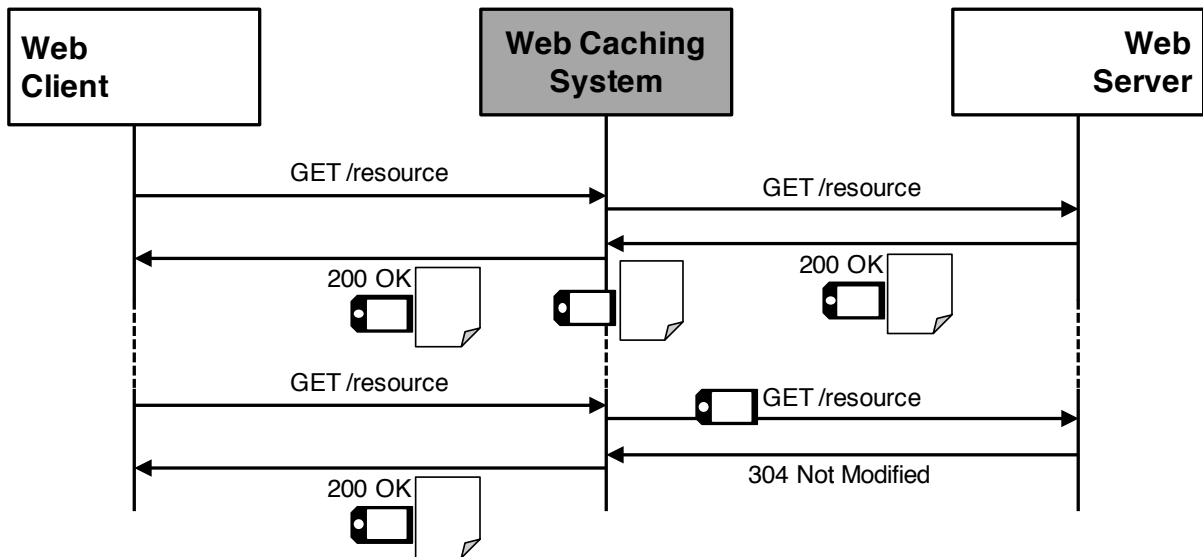


Figure 4.2: Functional principle of web caching with freshness validation. The response of the server contains meta data declaring that the freshness of the response needs to be validated.

tion requests based on a time-variant token require to append the `If-Modified-Since` header including either a self-defined timestamp or the timestamp value of the `Date` or `Last-Modified` header. A successful freshness validation requests results in a response with a `304 Not Modified` status code. Here, a cache returns a stored response to the client, but must update the header fields of the reused response with those of the response obtained for its validation request.

### Delivering Stale Resources

In certain circumstances, such as in cases of error or unavailability of the origin server, a cache may deliver stale responses instead of an error message. RFC 7234 allows to return stale responses, if the `Cache-Control` header of the resource in question does neither contain a `no-cache` nor a `must-revalidate` directive. If this is the case and a cache returns a stale response, it has to include the `Warning` header to the stale response informing the client about the expired content.

### Disabling Caching

If a origin server intends to prohibit all caches from storing a response, the endpoint must include `no-store` in the `Cache-Control` header field. To prevent shared caches from storing responses, origin servers have to add `private` to the `Cache-Control` header field. The `private` directive can also be used to prohibit shared caching systems from storing sensitive header fields such as `Set-Cookie`. Here, the `private` directive may contain values (e.g. `private="Set-Cookie, Server"`) in form of header field names declaring that a shared is allowed to store this response but must remove the header fields listed in the `private` directive.

## 4.2.2 Implicit Caching

If a response does not contain any explicit caching requirements declared by the `Cache-Control` or `Expires` header, a cache may either calculate an *implicit freshness lifetime* by using heuristics or perform a *implicit freshness validation*.

A cache may derive an expiration time based on the `Last-Modified` or `Date` header field if present. An implicit freshness lifetime can also be defined by using heuristics or predefined fixed value. RFC 7234 allows only to determine a implicit freshness lifetime if no explicit expiration time is set and distinct precondition are given. One requirement for calculating a heuristic freshness lifetime is a response with status code `200 Ok` to a GET request. Other status codes allowing the determination of implicit freshness lifetime can be found in Section 6.1 of RFC 7231 [FR14c].

Besides the determination of heuristic freshness lifetime values for responses without any explicit caching requirements, a cache may proactively initiate an implicit validation process if it notices that clients recurrently execute equivalent requests. This can be done by transforming the clients' requests to conditional requests containing the `If-None-Match` or `If-Unmodified-Since` header field. That is, the cache receives a request from the client which does not contain any conditions. The cache owns a suitable stored response which can be reused for satisfy the client's request. This cached response has neither any explicit caching requirements nor any explicit freshness lifetime definitions. The only cache-related information of this response is a validation token, i.e. the `ETag`, `Last-Modified` or `Date` header field. Instead of recycling this response directly, the intermediary proactively converts the client's request into conditional request including one of the validation tokens and send it to the server. Such a behavior is considered as implicit, since the cache performs a validation by it's own choice. The client and server do not tell the cache to do this. In contrast, an explicit validation means that the origin server forces the cache to perform an validation before reusing the stored response. In other words, the web caching system must perform a validation of the response.

## 4.2.3 Client-originated Caching Policies

RFC 7234 also defines cache control directives for the client. As with the cache, a client can include the `If-None-Match` or `If-Modified-Since` header field to the request for starting a conditional request. Also, a client may use the `max-age`, `no-store` or `no-cache` directives with the `Cache-Control` header field in a request for declaring explicit caching requirements. Additionally, with the `max-stale` directive a client can specify that it is willing to accept a request with a maximum staleness. The `min-fresh` parameter allows a client to declare that it wants a response that remains fresh for at least the defined number of seconds. If a client only wishes to obtain a stored response from a cache, it can include the `only-if-cached` directive in the `Cache-Control` header field. In this case, a cache can either return a stored response or it can reply with the status code `504 Gate Timeout`, if the cache does not possess a suitable response.

## 4.2.4 Defining new Cache Keys

The cache keys represent the indexes for mapping a request to stored response. By default, caches use the HTTP method and the URL as cache keys in order to determine whether it

possesses a valid response for the client. A server can also add further request header fields as additional cache keys. To do so, the service provider needs to include the `Vary` header field containing a comma separated list of header field names to be considered as additional cache keys (e.g. `Vary: Accept, Accept-Language`).

#### 4.2.5 Invalidation of Freshness

The invalidation of freshness defines an action which prohibits the reuse of responses even though its freshness lifetime has not expired. RFC 7234 forces caches to invalidate a response freshness if the result of a request to a recurring URL is not a error response and embodies an unsafe HTTP method including PUT, POST, PATCH or DELETE. The standard defines the status code classes 4xx and 5xx as errors.

#### 4.2.6 Partial Content

Beside ensuring scalability and performance, caching systems may include other services. The retrieval of a partial content of stored responses is, e.g., another core topic of RFC 7234 and can be used for optimizing the performance likewise. If the transfer of a response is interrupted for some reasons and a client has not received the whole content, it may perform a range request retrieving only the missing bytes of the response instead of invoking the whole transmission again. If a cache supports partial content retrieval, it can handle such requests without invoking the origin server.

#### 4.2.7 Security

The HTTP caching specification disallow a cache to store a response to a request containing the `Authorization` header field unless the origin server explicitly allows it with a corresponding header field value. A origin server can also prohibit a shared cache from storing a response by adding the the control directive `private` to the `Cache-Control` header. This header field only permit private caches to store and reuse the corresponding response. Moreover, the web caching standard discusses privacy issues and security issues such as knows attack vectors including cache poisoning which need to be considered by developers or providers applying caching.

### 4.3 Web Browser Caches

A conventional web browser contains multiple caching subsystems: a cache storing HTTP messages, the AppCache [W3C17] and the localStorage also known as Web Storage [Hic16]. The AppCache is used to store web resources but it is designed to cache web pages for offline usage. The to be cached files are specified by a manifest document and not by HTTP control directives. The localStorage is a key-value database which provides an Javascript API for the storage of data. As with AppCache it is not considered to store HTTP messages based on HTTP control directives. Note, that in this paper the AppCache as well as localStorage are out of scope. Therefore, all description related to caching in this paper only refers to the web browser cache



which stores HTTP messages based on HTTP control directives according to the specifications in RFC 7234.

According to a pilot study which we will discuss in more detail in Section 4.5, web browsers reuse stored responses from the internal caching systems when the request is executed via the XMLHttpRequest API [WHA18] and the HTML tags `<script>`, `<img>`, `<a>` or `<link>`. One exception where the web browser never utilizes the cache, is when a user explicitly reloads a web page with the corresponding buttons. In case of entering a URL in URL bar, the analyzed web browser however show an inhomogeneous behavior. While Firefox and Edge still strive to reuse cached content if fresh responses are available, Chrome and Safari always bypass the internal cache and fetch a new response. This diversity needs to be taken into account by web developers which consider caching in the business logic. To investigate further peculiarities and diversities of web browser caches, web developers require testing tools. Besides the detection of disparities, there is also a need for testing tools for identifying misbehavior of caching systems. Caches which do not work properly may induce malfunctions, leading to a loss of scalability, efficiency and performance. One intuitive example is when caches either ignore or miss freshness lifetime policies. That is, caches do not leverage possible performance gain as they forward each request to the origin server and do not store or reuse responses for recurring requests. A cache malfunction can also be the wrong reuse of responses for equivalent requests. This means a cache reuses stale content or response which is not returned if the client would have communicated with origin server directly.

## 4.4 Related Work

The misuse of complex systems often results in faulty software as described in Section 4.3. Especially in terms of Security API misuse the consequences can be severe [SB12; Fah+12; LG17]. In this context, the availability of meaningful tools and documentation is understood as crucial foundation for developers [GS16; GL16; Kru+17]. In this paper we aim at exploring the situation in respect to web caching.

As caching is a vital requirement for providing scalability and performance in web-based systems, a lot of tools and plugins evaluating web browser caching facets are available. The browser-native developer tools provide information on whether a resource is retrieved from the origin or the cache. Moreover, Firefox provides access to the HTTP responses stored in the cache when entering `about:cache` in the URL bar. Likewise, there exist several web browser plugins such as CacheViewer [Ben15] and ChromeCacheView [Nir18] which allow to inspect the cached web resources. These tools are only for analyzing what HTTP messages and how long the content is stored in web browser's cache. None of these approaches consider to investigate whether a cache stores and reuses HTTP messages in compliance with RFC 7234. The next section introduces such an approach for analyzing whether a cache behaves according to the requirements defined by RFC 7234.

## 4.5 Test Methodology

The main objective of our test methodology is to find a meaningful test suite containing a comprehensive collection of structured test cases for analyzing web caching systems in regards

to non-conformance, misconfiguration, malfunction and potential vulnerabilities. Note, that our methodology does not aim to evaluate the performance of web caching systems. Also, the aim of our approach is not to conduct a security analysis of web-based systems which support caching. For this purpose, penetration testing tools are more suitable. Likewise, our test suite is not considered to analyze whether a cache is compromised or already poisoned with malicious content.

The first step towards a systematic analysis of web browser caches is a deep understanding of the web caching foundations which is laid down in Section 4.2. This section covers and summarizes the core aspects of RFC 7234. Based on Section 4.2 we develop our test suite covering all aspects defined in the web caching foundations and works addressing web caching issues. The subsections of Section 4.2 build the structure and topics of our test suite as shown by the subsequent list.

1. Explicit caching controlled by server
2. Implicit caching
3. Client-originated caching policies
4. Cache key adaptation
5. Invalidation of freshness
6. Partial content
7. Security
8. Other

We use this structure as a baseline for inferring and classifying test cases in a methodical manner. To do so, we perform an in-depth review of RFC 7234 as well as available literature describing caching issues and extract cache requirements, cache behaviors as well as cache malfunctions. Based on this gathered information we infer test cases and group them into the topics of our test suite. With this strategy we ensure that our test suit covers all test cases for analyzing all requirements and cache behaviors described in RFC 7234. As the test case collection covers all facets of the web caching standard, malfunctions issues related to conformance disobey can be detected. On the basis of this methodology we were able to identify 397 test cases covering constructive as well as destructive tests.

In order to get a platform-independent test environment, we use XMLHttpRequests as all web browser provide a standardized Javascript API for issuing such requests. Also, the XMLHttpRequest API is the only platform-agnostic approach for executing the 397 test cases in an automatized manner. Tools like Puppeteer [Goo18] and Selenium [Sel18] could be used for web cache test. These instruments are, however, only implemented for desktop web browsers. Mobile browsers cannot be analyzed with these tools. Also, there is no evidence that the headless mode used by these tools behaves the same as using a web browser normally. Therefore, we use XMLHttpRequest to analyze the caching behavior, as this approach can be implemented and executed for all web browses including desktop and mobile. Another reason for the utilization of XMLHttpRequests is fact that the XMLHttpRequest API provides the option to set self-defined header fields. Adding header fields is, e.g., required to include requests header fields such as `Cache-Control` which are needed for analyzing client-originated caching policies.

To verify whether XMLHttpRequests can be utilized for obtaining a generally valid analysis of web browser caches, we conducted a pilot study with Chrome, Firefox, Safari and Edge. The study shows that all requests are satisfied by a fresh stored response unless the user explicitly wishes to retrieve a new response. That is, Chrome, Firefox, Safari and Edge check back the same internal cache when the request is executed via the XMLHttpRequest API and the HTML tags `<script>`, `<img>`, `<a>` or `<link>`. The web browsers only omit the cache when a user explicitly intends to update a web page by clicking the reload button or pushing the F5 key. Safari and Chrome also bypass the cache when a user enters a URL in the URL bar. Note that we conduct this study only for the desktop version of Chrome, Firefox, Safari and Edge, as we require to utilize the native developer tools to obtain a reliable analysis on whether a response is reused.

## 4.6 Implementation of Testing Tool

In order to make our collection of test cases easily adoptable we developed a test environment providing a simple test case syntax that allows to extend our base test suite in a straightforward manner. We provide the cache testing tool containing the 397 test cases as free software. It can be downloaded from <https://github.com/das-th-koeln/Cache-Testing-Tool>.

### 4.6.1 Web Browser Cache Testing Tool

Figure 4.3 shows the architecture of our cache testing system. We designed it as functional black box test with the test objective being the cache. The architecture consists of a cache testing server which is accessed and controlled by the cache testing client. Both endpoints need to be under control of the test environment. By constructing specific sequences of message flows, the behavior of the caching system can be examined.

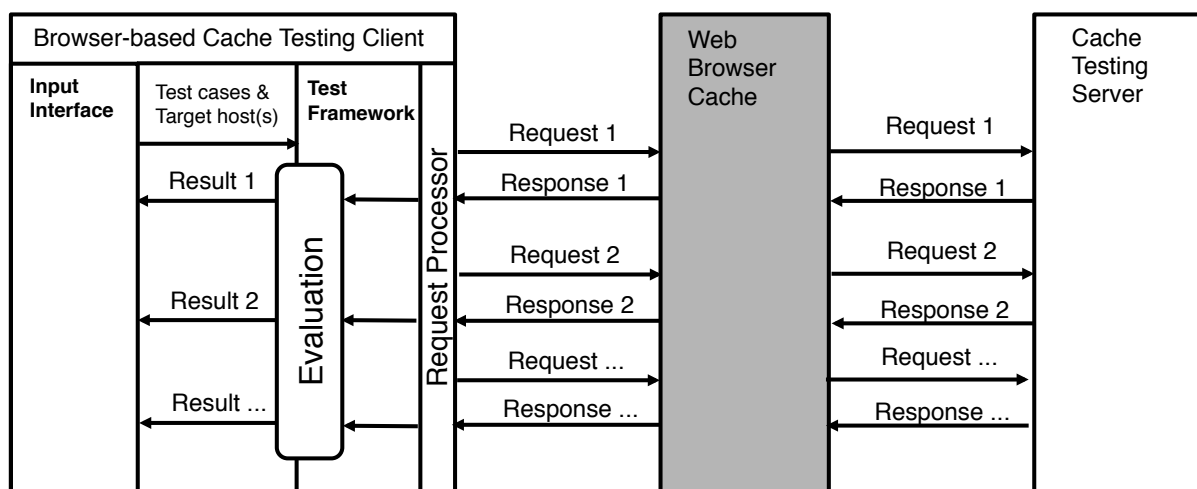


Figure 4.3: Architecture of the automated web browser cache testing tool

We realized the cache testing client as a web application which consists of an input interface where the test vectors can be defined and a Javascript-based test framework which executes the requests as well as evaluates the responses. The input interface requires two mandatory input

arguments: the host of the cache testing server and a set of test cases defined by our test case syntax that is introduced in the following section.

The test procedure starts by transferring the provided test cases to the test framework, where each test case is processed sequentially and converted into a corresponding request. This is done by a request processor inside the framework. This request processor converts the request/response definitions to requests according to test case syntax and sends them to the cache testing server. The requests contain specific meta information in order to unambiguously distinguish whether a response is originated by the tested cache or the server. Moreover, the test case syntax includes an option to define the expected response—e.g. in terms of RFC 7234. By this, the test framework can compare the obtained response with the expected response to automatically generate a test report.

#### 4.6.2 Test Case Syntax

The syntax of our proposed test case definition language is based on the command line tool *curl* [Ste18]. Listing 4.1 shows an example. Each test case contains a descriptive title and is composed of one or multiple commands. The statements are executed in the specified sequence. Each command represents one request and its expected response. Mandatory elements of each command are the HTTP request method and the URL of the targeted resource. By means of optional parameter, various extensions to the issued request and expected response can be crafted. This is mainly related to the HTTP headers. One remarkable parameter is the flag to specify the expected response. This can be adopted to obtain automated reports showing deviations from the defined baseline.

```
1 ## Descriptive title of test case
2 GET /rsc -c 'Accept:application/json;Cache-Control:no-store'
3           -s 'Cache-Control:max-age=10'
4           -p 5
5           -e 'ch:false;st:200'
6 GET /rsc -c 'Accept:application/json'
7           -s 'Cache-Control:max-age=10'
8           -p 10
9           -e 'ch:true;st:200'
```

Listing 4.1: Example cache test case defined using the introduced test case specification syntax

The example test case giving in Listing 4.1 specifies two commands. The first one is defined by the lines 2 to 5 and the second one by the lines 6 to 9. Both specified statements issue GET requests and provide the targeted resource URL without the scheme and host. The test case syntax allows to add one or more header fields to a request by the `-c` parameter flag. The first command in the example adds two header fields to the request: (1) the `Accept` header field with the value `application/json` and (2) the `Cache-Control` header field containing the value `no-store`. The header fields are separated from each other using the semicolon character as delimiter. Accordingly, by means of the `-s` parameter flag a set of header fields can be specified that are added by the cache testing server to the generated response. The `-p` parameter flag defines a pause in seconds which has to be waited before the testing client executes the next command. Breaks between requests are crucial for analyzing the freshness lifetime of a response for instance. Additionally, the `-e` parameter flag describes the to be expected properties of the response. These expected properties can be either requirements of the RFC 7234 or requirements of the own caching policy. If the resulting responses fulfill

all expected properties, then the test case is considered as successful. If a response does not embody one of the properties, than the web caching system may have a malfunction. In the first request/response definition, the first expected property `ch : false` (see line 5) assumes that the result is a fresh response created by the origin server and not a replayed one from the cache. The second expected property of the received response requires that its status code is `200 Ok` (expressed by `st : 200`).

Request	Response	
<pre>GET /rsc HTTP/1.1 Host: cache.example.org Accept: application/json Cache-Control: no-store X-Response: Cache-Control:max-age=10 X-Id: 123</pre>	<pre>HTTP/1.1 200 Ok Cache-Control: max-age=10 Content-Length: 11 Content-Type: application/json Date: Tue, 27 Feb 2018 13:29:28 GMT X-Id: 123  {"Id": "123"}</pre>	✓
5 seconds later		
<pre>GET /rsc HTTP/1.1 Host: cache.example.org Accept: application/json X-Response: Cache-Control:max-age=10 X-Id: 345</pre>	<pre>HTTP/1.1 200 Ok Cache-Control: max-age=10 Content-Length: 11 Content-Type: application/json Date: Tue, 27 Feb 2018 13:29:28 GMT X-Id: 123  {"Id": "123"}</pre>	✓

Table 4.1: Requests and responses produced by the example test case in Listing 4.1 while executed by the web browser cache test system

Table 4.1 depicts the request/response flow executed by the cache testing tool following the example test case specifications shown in Listing 4.1. The first request contains the GET method and targets the URL path `/rsc`. It includes the `Accept` and `Cache-Control` header fields asking to retrieve the addressed resource in `application/json` that must not be a stored copy from a cache. The header fields provided with the `-s` parameter flag should be added to the response and are therefore transferred to the cache test server in the `X-Response` header field of the request. If a request reaches and is processed by the cache test server, it appends the header fields contained in `X-Response` header field of the request to the response. In the given example, the server includes the `Cache-Control` header field with a value of `max-age=10`. Moreover, all requests comprise the `X-Id` header field which embodies a unique random id. The cache testing server adds the request id in the `X-Id` header field of the generated response as well as the response body (see response in row 2 of Table 4.1). The request id inside the request and the response allows to distinguish whether a received response has been issued by a caching system or the cache testing server. By means of the added `X-Id` header fields, one can assess that the first response is sourced from the server as it contains the same `X-Id` header field value as the triggering request. The second response instead is sourced from a cache (visualized by the gray background), since it comprises the request id of the first request and not the one from the triggering request. Finally, the cache testing server adds the `Date` header field to the response representing the response creation time. Table 4.1 shows that both responses fulfill the expected properties specified the statements in Listing 4.1. The first response comes from the server and contains the status code `200 Ok` while the second is recycled by the web caching systems and comprises the status code `200 Ok` likewise.

The whole test suite containing all 397 identified and specified test cases by means of the proposed test methodology can be obtained from <https://github.com/das-th-koeln/Cache-Testing-Tool>.

## 4.7 Empirical Study

With the introduced automated web browser cache testing tool, we analyzed the client-internal web browser caches of Chrome/Android v64, Chrome/Windows v64, Safari/Mac v11, Safari/iPad v11, Edge/Windows v16 and Firefox/Windows v58.

Table 4.2 summarizes the most significant results. The full anonymized analysis results can be accessed at <https://cachetester.github.io/cachetest/?cache=browsercaches>.

	Chrome/Android	Chrome/Windows	Safari/Mac	Safari/iPad	Edge/Windows	Firefox/Windows
<b>Explicit Freshness Lifetime</b>						
max-age	●	●	●	●	●	●
Expires	●	●	●	●	●	●
max-age & Expires	●	●	●	●	●	●
<b>Implicit Freshness Lifetime</b>						
Last-Modified	●	●	●	●	●	●
Date	○	○	○	○	○	○
<b>Explicit Freshness Validation</b>						
ETag & no-cache	●	●	●	●	●	●
Last-Modified & no-cache	●	●	●	●	●	●
Date & no-cache	○	○	○	○	○	○
<b>Implicit Freshness Validation</b>						
ETag	●	●	●	●	●	●
Last-Modified	○	○	○	○	○	○
Date	○	○	○	○	○	○
<b>Invalidation</b>						
PUT	●	●	●	●	●	●
DELETE	●	●	●	●	○	○
POST	●	●	●	●	●	●
PATCH	○	○	●	●	●	●
<b>Client-originated Policies</b>						
max-age	●	●	○	○	○	○
max-stale	○	○	●	●	○	○
no-store	○	○	○	○	●	●
min-fresh	○	○	○	○	○	○
only-if-cached	○	○	○	○	○	○
no-cache	○	○	●	●	●	●

Legend: ● RFC 7432 compliant, ○ none RFC 7432 compliant, ◐ partially RFC 7432 compliant

Table 4.2: Results of the empirical analysis of web browser caches obtained by the automated evaluation of 397 test cases focusing on the compliance to RFC 7432

Our study shows that all web browser caches comply with the freshness lifetime control directives `max-age` and `Expires`. Also, `max-age` is preferred over the `date` value of the `Expires` header field, if both definitions are present simultaneously. Web browsers also prefer the explicit cache requirements over the implicit ones. Moreover, all web browsers calculate an implicit freshness lifetime when a request contains a `Last-Modified` header field without any freshness lifetime control directives. However, an implicit freshness lifetime is not calculated based on the `Date` header field. As all web browser comply with all explicit freshness lifetime requirements of RFC 7234, this proper caching behavior has no negative consequences in terms of performance, security and privacy. The omission of the `Date` header for defining a implicit freshness lifetime may impair the communication performance, as this parameter is not utilized by the cache to save requests. Therefore, web developers must consider that the analyzed web browsers only use the `Last-Modified` header for specifying a implicit freshness lifetime.

In terms of freshness validation, all web browser caches perform an implicit validation if a response contains the `ETag` header field without any freshness lifetime header fields. An implicit conditional request is not executed when a request include a `Last-Modified` header field, as this date value is used by all browsers to define an implicit freshness lifetime. The `Last-Modified` header field is only used to perform a validation if the origin server requires to do it. This means a validation request with the time-variant validation of the `Last-Modified` header field is only done, if the origin server includes the `Cache-Control` header field containing the `no-cache` directive. The analyzed web browser



caches also initiate an explicit conditional request if the response contains the `ETag` and `Cache-Control` header field including the `no-cache` directive. Although, all responses contain the `Date` header field, which can also be used as validation token, none of the web browsers perform a validation on it. Instead, they start a request without any condition to retrieve a full response. The freshness validation with a conditional request instead of retrieving the same full response with body on each request saves a lot of network traffic when used properly. Web developers must be aware that the evaluated web browsers only use the `Last-Modified` or the `ETag` header to perform a conditional request. The `Date` header field or a self-defined timestamp is not considered.

Discrepancies have also been found out by the invalidation of response's freshness. Edge does not perform an invalidation for requests containing the `DELETE` method. This leads to Edge returning cached responses although they do not exist anymore on the origin server. Chrome does not consider `PATCH` as method for triggering an invalidation. Another noteworthy finding is that Chrome and Safari are the only web browser which do not trigger an invalidation, if the response resulting from a request with a supported unsafe method comprises an error code. Both web browsers still return a cached response in this case. This behavior is in conformance with the RFC 7234. Firefox and Edge, however, invalidate a response's freshness even though the result from a request with a unsafe method is unsuccessful. Ignoring unsafe methods as indicator for invalidating a response freshness leads to misbehavior in the business logic. Resources which have been removed or changed are not updated by the cache inducing that users obtain stale responses. Especially, REST-based web services [RR08] are affected by this malfunction.

The client-originated caching policies and control directives are only partially supported by the analyzed web browsers. Chrome does not support the client-side `no-cache` control directive, as it does not perform a validation request when it is present. Instead Chrome initiates a request without any condition causing the transmission of the full response. The `no-store` directive is only honored by Edge and Firefox. The analyzed web browsers also behave differently, when a request contains the `Cache-Control` header field with `max-age` directive. Chrome and Firefox return a response according the `max-age` requirements of the client, while the others ignore this header field value. This inhomogeneity is also shown with the `min-fresh` and `max-age` directives. Firefox is the only browser which supports `min-fresh` and `max-stale`. Safari honors `max-stale` but ignores `min-fresh`. Chrome and Edge neglect both. None of the tested browsers support the `only-if-cached` directive. Not honoring the explicit caching requirement of the request may induce the same consequences as not considering the server-side ones. On the one hand, refusing the request's caching requirements in some cases is crucial for retaining the scalability. The client can use the `no-store` or `max-age=0` directive to force the cache to fetch each response from origin server without considering stored contents. If a web browser supports both control directives, these requests may reduce the scalability and increase the workload of the origin server as each request is forwarded to the endpoint. To maintain the scalability, a cache can simply ignore and disobey cache-related client header fields. On the other hand, disrespecting client-side control directives which require to fetch a response from the origin server can lead to users becoming victims of cache poisoning attacks such as the threats presented by Jia et al. [Jia+15]. Client intending to circumvent this attack must add `no-store` or `max-age=0` to the `Cache-Control` header field which mandate the cache to fetch a new response from the origin server. Such a circumvention is only possible, if web caching systems support and honor the control directives of the client.



## 4.8 Conclusion

Web caching systems are an important intermediate component of contemporary distributed systems. The knowledge of and compliance with caching standards are crucial prerequisite in order to achieve high scalability and performance levels. The lack of informed decision making by web developers can lead to more severe issues than a simple degradations in efficiency, though, including privacy and security related concerns.

This paper provides a first systematic study of web browser caches. Our findings provide insights on their behavior and conformance with relevant standards. The empirical analysis reveals compliance discrepancies and diversities across web browser caches which need to be carefully considered by developers. These uncovered non-conformances and their accompanying side effects emphasize the need for meaningful tools for exploring the reliability and particularities of web browser caches. Such a tooling has been proposed. Although this paper solely focuses on web browser caches, the introduced cache testing approach can be generalized to analyze other types of caches including e.g. proxies and CDNs. Also, many other stakeholders besides web developers might benefit from the proposed cache testing tool including administrators, test engineers, software operators, DevOps teams, researchers as well as developers of caching systems and web browser vendors. The presented cache testing systems including the whole test suite is available for download at <https://github.com/das-th-koeln/Cache-Testing-Tool>.

Further work will analyze other web caching systems and other web caching facets in respect to security services. The aim is to propose robust protection means which do not impair the scalability and performance of web-based systems.

# Chapter 5

## Mind the Cache: Large-Scale Explorative Study of Web Caching

### Summary of this publication

**Citation** H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Mind the Cache: Large-Scale Analysis of Web Caching*. In: *34rd ACM/SIGAPP Symposium on Applied Computing (SAC)*. 2019. URL: <https://doi.org/10.1145/3297280.3297526>

**Status of Paper** Published

**Type of Paper** Research Paper (Conference)

**Ranking** GGS: A-, CORE: B, LiveSHINE: A, Microsoft Academics: A

**Aim** This paper aims to study the compliance of proxy caches and Content Delivery Networks (CDNs) based on the testing framework in Paper 3 (Chapter 4).

**Methodology** We use the test suite and caching testing tool of Paper 3 (Chapter 4) to analyze the web caches. Moreover, we extend additional test cases with the test specification language.

**Contribution** The analysis found out that the analyzed caches do behave differently in many respect, not always conforming with the respective standardization. Some observed peculiarities do even have the potential for future incidents and they remain unnoticed by system integrators and administrators without proper tooling.

**Co-authors' contribution** See Paper 4 in Section 1.1.1.

## 5.1 Introduction

As the world becomes more and more digitized, the interconnected software increases in size often forming so-called ultra-large scale systems [Fei+06]. Caching plays a central role to enable and ensure the scalability and performance of such systems. From an engineering viewpoint this puts new demands on software developers, as they have to deal with caching components in their systems.

At present, the web is arguably the world’s largest distributed system [Fie00]. The intense use of caching is one main reason for its growth [Wan99; BO00a]. A web cache represents a subsystem for coordinating the transparent storage and retrieval of recyclable HTTP responses [FNR14]. By this, a web caching system potentially reduces three quantities: the number of requests that reach the origin server, the volume of network traffic resulting from document requests, and the latency that an end-user experiences in retrieving a document. Moreover, when serving recurring requests on behalf of an origin server that is not online for some reasons, caches contribute to an increased availability of web-based services. In terms of security, caches do have two faces, though. One is, that a decentralized scattering of multiple caches to distinct network regions renders distributed denial of service attacks much more cumbersome and ineffective, since such attacks will be out-weighted by the distributed caches. On the other side, connections protected by transport security means such as transport layer security (TLS) [Res18] need to be terminated by the caches in order to fulfill their duties. This breaks the confidentiality of the data exchange.

The knowledge of and compliance with relevant caching standards and technologies are crucial in many respects. Ignoring them affects scalability and performance benchmarks in the first place. Non-adherence may induce further impacts, including mission critical side effects in the application behavior, as it has been demonstrated, e.g., by the web cache deception attack [Gil17]. Here, sensitive account information—not intended for caching—has been publicly accessible via a cache. This vulnerability exploited a combination of a malfunctioning cache and faulty web application. This example emphasizes the potential impacts of improper caching in distributed systems. As the foundations of the web are the driving forces for many contemporary software paradigms including e.g. SOA [Erl07], Cloud [EPM13b], REST [Fie00] and Microservices [New15], the increasing general relevance of web caching in software engineering becomes evident.

To prevent issues stemming from inappropriate web caching, one need to have a deep understanding of the current state of play. In Section 5.2 we therefore briefly recap key aspects of web caching. Based on these foundations, in Section 5.3 we discuss consequences of caching misbehavior and misuse respectively. With the aim to mitigate caching-induced malfunctions in distributed software systems, we mandate for proper test tools that are currently lacking, as the related work review in Section 5.4 manifests. In Section 5.5, we introduce our shared web cache testing tool. By means of the introduced testing tool, we conduct an empirical study of six proxy caches and one CDN with the purpose of evaluating the introduced approach and obtain a systematic analysis of available shared caching systems. The main results are discussed in Section 5.6. Overall, they do affirm the relevance of appropriate web cache testing tools to support stakeholders interacting with web caches such as developers, integrators, administrators and researchers. In Section 5.7 we conclude with an outlook on challenges in web caching putting a special emphasize on security-related considerations.

## 5.2 Web Caching

The caching of frequently used web resources in order to reduce network traffic and optimize application performance is one of the main reasons of success of the web [Wan99]. Many classes of web caching systems evolved over time and are utilized in various locations on the path between client and server (see Figure 5.1). One major distinction point of caching systems beside the location is the differentiation between *private* and *shared* caches. A *private cache* stores and reuses stored responses only for one single user while a *shared cache* can return stored responses for multiple users. Typical private caching systems are client-internal caches. The web browser cache is a prominent example here. Backbone caches including e.g. content distribution networks (CDNs) and client-side forward proxies as well as server-side reverse proxy caches are usually utilized as shared caches as they replay stored resources for multiple web clients. Web frameworks or content management systems often provide server-internal caching systems that can be implemented as private or shared cache. For instance, the WordPress plugin WP Super Cache<sup>1</sup> or the Java-based cache Ehcache<sup>2</sup> are able to store and reuse (dynamically generated) contents. These types of caches can usually be used to serve multiple users as well as a particular user.

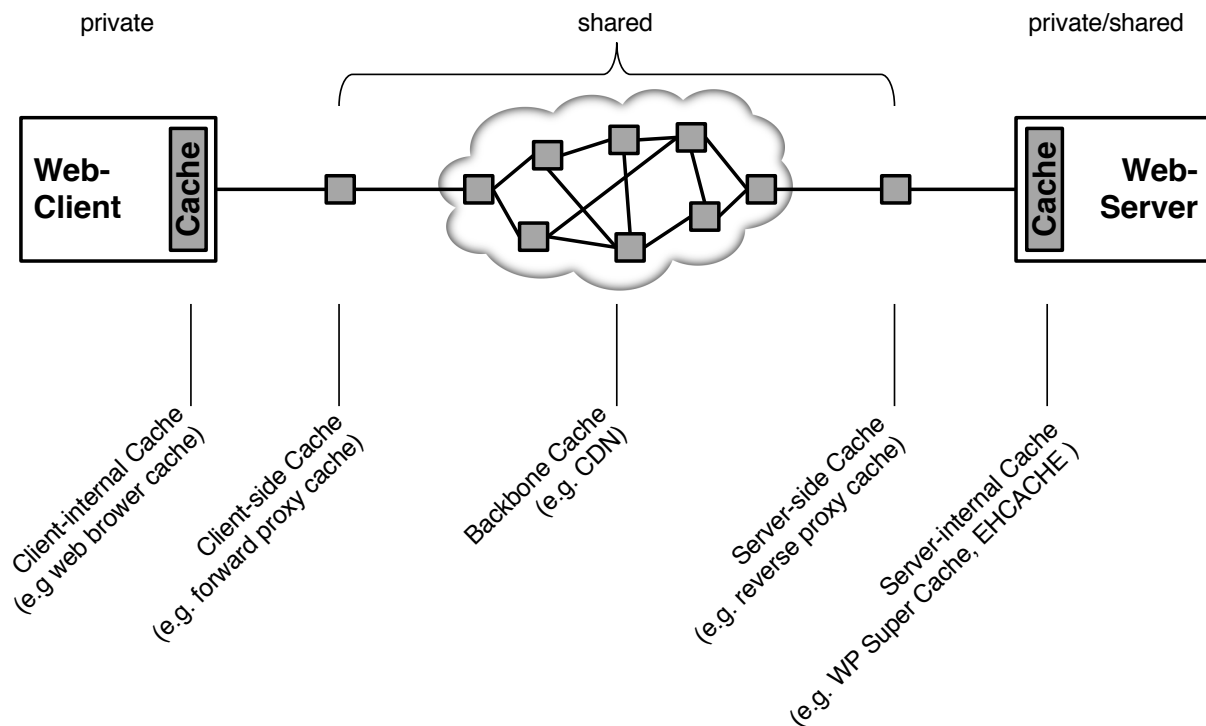


Figure 5.1: Different types of web caching systems classified by location and resource access policy

The reliability of private and shared caches in terms of proper exchanging and storing content is essential for a dependable caching environment. Therefore, all types of web caching systems are required to ensure that clients are always provided with fresh and authentic copies of the original response. To do so, the RFC 7234 [FNR14] defines policies and cache control mechanisms for origin servers, web caching system vendors and clients.

1. <https://wordpress.org/plugins/wp-super-cache/>

2. <http://www.ehcache.org/>

### 5.2.1 Freshness

How a particular response is cached is mainly governed by the origin of the resource, the server. It decides whether a resource is cacheable at all and if so, under which circumstances it can be delivered by a caching intermediate. To do so, the server explicitly expresses freshness properties within the concerning response (see Section 5.2.1). In cases, in which the server did not specify any caching requirements, the caching intermediate can assign policies by its own (see Section 5.2.1).

#### Explicit Caching

For origin servers, there are two general approaches for explicitly ensuring the freshness of a cached response. The first approach is to define an *explicit freshness lifetime*. The other strategy is the *explicit freshness validation* also known as *explicit conditional request*.

An explicit freshness lifetime defines the time span, which allows a cache to reuse a stored response. Here, a cache is able to satisfy recurring requests with the stored content without requiring any server intervention. There are two control directives for defining the explicit freshness lifetime. One is the `Expires` header which contains the absolute date of expiration. The other is the `max-age` directive which includes the relative expiration time in seconds. The `max-age` directive is included in the `Cache-Control` header. For instance, `Cache-Control: max-age=60` indicates that the corresponding response can be considered as fresh for the next 60 seconds. Both control directives are valid for private and shared caches. The `s-maxage` keyword indicates the same instructions as `max-age`. The only difference is that `s-maxage` is exclusively dedicated for shared caches. A cache which reuses a stored copy based on the freshness lifetime must include the `Age` header to the replayed response. The `Age` header indicates number of seconds since the response has been received by the cache.

Another option to ensure the freshness is to validate the stored content with a conditional request. In this case, the cache may own a suitable response. Still, it has to check back each recurring request with the server in order to determine whether the stored response is still fresh. If the validation request is successful, meaning that the stored content in the cache has not changed in the meantime, the server replies with a response containing the status code `304 Not Modified`. Note that this response does not include any content in the body. If a cache receives such a message, it can replay the cached content to the client, but it must update the header of the stored response with the header of the `304 Not Modified` response. If the validation request is unsuccessful, meaning that the requested resource has changed, the server must return the `200 Ok` status code containing the updated resource in the body. A cache receiving this message must forward the response to the client. Simultaneously, it replaces the stale response with the fresh one.

If a server intends to force a cache to validate the freshness of a stored content, it adds the `Cache-Control` header to the response containing one of these values: `no-cache`, `must-revalidate` or `proxy-revalidate`. The `no-cache` directive forces the cache to validate the response's freshness for each recurring request to the same resource while `must-revalidate` only requires to verify the response when the freshness lifetime is expired. Both instructions must be considered by shared as well as private caches. `proxy-revalidate`

has the same functionality as `must-revalidate`, with the difference that it is only dedicated for shared caches.

A conditional request always contains the `If-None-Match` or `If-Modified-Since` header including a validation token as value. RFC 7234 defines two types of validation tokens: an opaque entity tag and a time-variant parameter. The `If-None-Match` header must contain one or multiple opaque validation tokens which are included in the `ETag` header of the cached response. Conditional requests with the `If-Modified-Since` header contain an absolute date which can be obtained from the `Last-Modified` or `Date` response header.

In some cases, a cache can also return a stale response to the client, if the server is not available for any reasons. A stale response must include the `Warning` header informing the client about the expired response. Caches are not allowed to replay a stale response if the expired stored `Cache-Control` response header contains at least one of the following keywords: `no-cache`, `must-revalidate` or `proxy-revalidate`.

Servers can also prevent caches from storing and reusing particular contents. Here, endpoints must include `no-store` or `max-age=0` to the `Cache-Control` response header. The web caching standard also provides the option to prevent shared caches from storing certain responses. To do so, content providers must include `private` to the `Cache-Control` response header. From the security and privacy viewpoint, this control directive is essential as it allows content providers to disallow shared caches from storing sensitive information. `private` can also be utilized for prohibiting shared response from reusing and storing sensitive header fields such as `Set-Cookie`. For instance, if `Cache-Control: private="Set-Cookie"` is appended to a response, a cache can still reuse this message for recurring requests, but it must remove the `Set-Cookie` header.

## Implicit Caching

RFC 7234 allows caches to define an *implicit freshness lifetime*, if a response does not contain any explicit caching requirements. An implicit freshness lifetime can be defined by a fixed value configured by a cache administrator or a heuristic algorithm can derive it. If a response contains the `Last-Modified` or the `Date` header, a cache can also use the time-variant value within these header fields to derive an implicit freshness lifetime. If a cached response is delivered from the cache due to an implicit freshness lifetime, no further validation takes place with the server. The cached resource contains the `Age` header signaling that a reuse happened without server intervention.

Another cache-controlled initiative in the absence of any explicit caching is the *implicit validation*. The procedure of an implicit validation is the same as for an explicit validation. In case time-variant tokens within the `Date` or `Last-Modified` headers are present, the cache can initiate an *implicit freshness validation* request towards the server. Similarly, caches can also initiate an *implicit conditional* request in case opaque tokens in the `ETag` header value are at hand.

### 5.2.2 Client-originated Policies

As we have seen, the majority of the caching directives are dedicated to the server and do henceforth reside in HTTP response messages. For some use cases, however, also the client

can benefit from expressing caching requirements. RFC 7234 therefore also provides client-side control directives which are declared in HTTP request messages. If a client maintains an internal cache subsystem, it can use the `If-None-Match` or `If-Modified-Since` header to issue a conditional request. Moreover, a client may include a `max-age` directive in the `Cache-Control` request header for declaring that it only wishes to retrieve a stored response of a maximum age. If a cache contains a suitable response, which does not exceed the specified maximum age, it replays the stored content to the client. If the cache does only have a stale copy, it must forward the request to the server to retrieve a new fresh response. The control directives `no-store` and `no-cache` of the client-side `Cache-Control` header implies same instructions as for the server-side counterpart.

The `min-fresh`, `max-stale` and `only-if-cached` control directives are exclusively defined for the `Cache-Control` request header. With `min-fresh=60`, e.g., a client declares that it only wishes to retrieve a cached copy if the corresponding stored response is still fresh in the next 60 seconds. The `max-stale` directive implies that client is willing to accept a stale response whose expiration time does not exceed the specified number of seconds. If a client only wishes to retrieve a cached response, it can use `only-if-cached`. Caches which are not able to return a cached response for such a request must return the status code 504 `Gateway Timeout`.

### 5.2.3 Cache Key Adaption

The cache key unambiguously identifies a cached resource. By default, the cache key consists of the HTTP method and the URL contained in the request. The corresponding response is then identified by this attribute combination. Content providers intending to extend the cache key can do so by specifying the according request header names to the `Vary` response header. If a response contains, e.g., the header entry `Vary: Accept, Accept-Encoding`, in addition to the HTTP Method and the URL the `Accept` and `Accept-Encoding` headers are part of the cache key.

### 5.2.4 Invalidation of Freshness

Invalidation of freshness is the process that cancels the freshness lifetime of a cached response, even if the stored content has not expired yet. According to RFC 7234, a cache must invalidate a freshness lifetime of a response if the result of a request to this response contains a unsafe method and is not a error message. HTTP standards define `PUT`, `POST`, `PATCH` and `DELETE` as unsafe methods, since these action change the state of a resource. Error messages are responses with the status code classes `4xx` and `5xx`.

### 5.2.5 Partial Content

Another optional feature of a cache is allowing the client to retrieve partial content. Requests intending to retrieve a partial body are considered as range requests. These requests are useful in case the communication between client and cache is interrupted for some reasons. Instead of requesting the whole response again, a client can perform a range request retrieving only the missing parts.



## 5.2.6 Security

Caching systems provide scalability, performance and availability. Distributed caching systems such as CDNs can provide an additional protection mean in terms of DDoS attack prevention. However, using a cache may also open the door for security issues. Therefore, RFC 7234 specifies security considerations for cache vendors and applications using caching systems. One important recommendation of the web caching standard is to prohibit the storage of responses which results from a request containing the `Authorization` header, unless the server allows it with a explicit freshness lifetime header value. This rule prevents caches from storing and reusing responses that require an authentication. Moreover, RFC 7234 suggests implementation hints for thwarting cache poisoning attacks. The consequences of cache poisoning attacks and other cache malfunctions will be discussed in the next section.

## 5.3 Consequences of Malfunctioning Caching

Caches that do not function properly may lead to the loss of the desired properties in terms of scalability and performance. One intuitive example is when either caches ignore or miss freshness lifetime policies. That is, caches do not improve the performance as they forward each request to the origin server and do not maintain response copies for recurring requests. Cache poisoning attacks are another threat which can adversely affect the reliability of caches.

The *request smuggling attack* [Lin+05] is a shared cache vulnerability which executes a request including two `Content-Length` headers. Even though the handling of such requests is prohibited by the HTTP standard, their presence may confuse a cache, so that a malicious response can be injected to the intermediary. This injected response is then reused by the shared cache for recurring requests.

The *host of troubles* attack [Che+16] is another cache poisoning threat which exploit the presence of two equal headers. In this attack, a request containing two `Host` headers fools a cache to store a response under the cache key of the injected `Host` header.

The *web deception attack* [Gil17] shows a misbehavior of caching systems which also results from a violation of the RFC 7234. Here, the cache still stores and recycles responses containing sensitive account information even though the control directives prohibits it. In conjunction with a flaw in the requests routing process, this attack allows to access account information of other users.

All above-mentioned malfunctions arise mainly from dishonor of HTTP standards. Triukose et al. [TAR09] show a vulnerability which does not result from a non-conformance with the HTTP standards. This attack utilizes the fact that the web caching standard defines the HTTP method and URL as default cache keys. Hence, each change in a URL including the query part—i.e. <http://example.org?<query>>—forms a new cache key. This means, a cache must forward each request, which includes a URL with the same scheme, host and path but a different query if the intermediary does not possess a suitable stored response. The presented attack exploits this definition by using a CDN for initiating a DDoS attack. Even though service providers utilize a CDN to hamper such attacks, the authors demonstrate that adding a new random string to query part of the URL prompts always an edge server within a CDN to forward this request to the origin server. The authors make use of this behavior and perform requests with different random query strings to all edge servers. As the edge servers conform to RFC 7234 and ignore the

fact that the origin server produces the same content independent from the query part, the huge amount of requests produced by the edge servers exceeds the capacity of the origin server. This hence causes a denial of service due to generated workload. In this case, the distributed web caching systems are in compliance with RFC 7234 and forward each request, as the URLs are different. However, the cache does not know that the origin server considers requests including URLs with the same scheme, host and path but a different query as equivalent requests.

This discussion already emphasize that the reasons for cache malfunctions are manifold reaching from error-prone configurations, missing input validations to non-compliance with RFC 7234 and other relevant standards. Also, relying on the conformance with HTTP standards alone does not mean that scalability and reliable caching in particular is ensured in all cases as shown by Triukose et al. [TAR09]. Moreover, caching policies need to be adjusted and tested according to the requirements and properties of the web application's business logic in addition. In order to mitigate and detect malfunctions in caching systems with the aim to optimize the scalability, performance and protection of web applications there is a need for test methodologies and tools analyzing the compliance and reliability of web caching systems.

## 5.4 Related Work

The previous section showed that programming and configuration mistakes in distributed software systems layered by many distinct components can lead to severe consequences. To mitigate such misuses, software developers and administrators requires meaningful auxiliaries in terms of testing tools and documentation [GS16; GL16].

Many available cache testing tools are used for performance analysis such as Web Polygraph [Web18] or Fiddler [Tel18]. Cache performance studies and performance optimization approaches are also subject of much research work such as [Car+05; BVR13; RLB03].

As we have seen, scalability and performance are not the only relevant metrics to evaluate in the context of web caching. Besides this, there are also many publications and tools in respect to vulnerability testing. Jia et al. [Jia+15] conducted a study on web browser cache poisoning (BCP) attacks. The authors investigate the feasibility of BCP attacks in various popular desktop browsers. Based on these findings, they propose guidelines for users and browser vendors to mitigate such threats. Also, James Kettle presented a set of cache poisoning attacks on CDNs and proxies which does not result from a vulnerability in the web caching system itself [Ket18c]. Unlike the threats described in Section 5.3 where an attacker exploits a flaw in corresponding caching systems, the cache poisoning attacks presented by James Kettle are based on a vulnerability in a web framework or content management system. Moreover, there are a lot of penetration testing tools, e.g. OWASP Zap [BPM18] and Burp Suite [Por18], which are utilized for detecting vulnerabilities in web-based systems which may include a cache. In summary, the mentioned works on caching focus on two major topics: (i) performance optimization and (ii) vulnerability analysis.

Studies on the compliance with web caching standards have not been on the focus of academic work so far. The Co-Advisor project [The18] has been one of the first projects targeting HTTP compliance testing. However, this product only provides a set of test cases for free. Users have to buy additional test cases. Also, it does not allow the user to define custom test cases. Users requiring customized test cases need to pass their test specification to the Co-Advisor vendor. Hence, defining test cases for investigating own cache configurations or caching aspects that are

not predefined by the Co-Advisor project is not feasible in a straightforward manner. Moreover, the documentation of Co-Advisor project only refers to RFC 2616 [Fie+99] which is an obsolete HTTP standard. There are also two additional projects from Achintha Reemal [Ree15] and the HTTP2 specification working group [IET14] focusing on HTTP compliance testing. As with the Co-Advisor project, both tools do not allow the user to define own test cases without modifying the source code. Nguyen et al. [NLF18] present a methodology and cache testing tool which have been used to conduct an empirical analysis of web browser caches. Their approach allows customizing and extending further test cases in a simple manner. However, their tool covers the client-internal caches inside web browsers only. In this work, we extend their approach by implementing a cache testing tool for shared caches. We aim to use this tool for analyzing popular proxy caches and CDNs in a systematically manner.

## 5.5 Cache Testing Tool

Our approach is based on the methodology of Nguyen et al. [NLF18] who perform an empirical study of web browser caches. The first goal of this methodology was to define a comprehensive test structure, which allows to derive a considerable test suite for analyzing the reliability of web caching systems. To do so, Nguyen et al. conducted an in-depth review of RFC 7234 as well as publications related to HTTP and web caching. As a result they proposed a test suite structure with the following topics:

1. Explicit caching controlled by server
2. Implicit caching controlled by cache
3. Client-originated caching policies
4. Cache key adaptation
5. Invalidation of freshness
6. Partial content
7. Security
8. Other

On the basis of this methodology and a proposed test case specification language Nguyen et al. were able to identify 397 test cases covering constructive as well as destructive tests. Even though their paper covers web browser caches only, the test collection can also be utilized for analyzing all kinds of web caching systems. In this paper, we utilize their methodology, test suite and test case specification language to build a web cache testing tool for investigating proxy caches and CDNs.

### 5.5.1 Architecture

Figure 5.2 shows the architecture of our web cache testing system which is based on the approach of [NLF18]. We implement our tool as functional black box test with the main test objective being the cache. The architecture of the tool consists of a test server, which is accessed and controlled by the test client. Both endpoints need to be under control of the test environment.

The proposed architecture enables to explore forward or reverse proxies and CDNs. The server-internal caches WP Super Cache and Ehcache cannot be tested by our system, though, as these web caching systems are different to others. Here, the caching systems are not designed to store responses according to protocol header elements of HTTP messages. Instead, the caching behavior is set via internal elements of the corresponding programming environment.

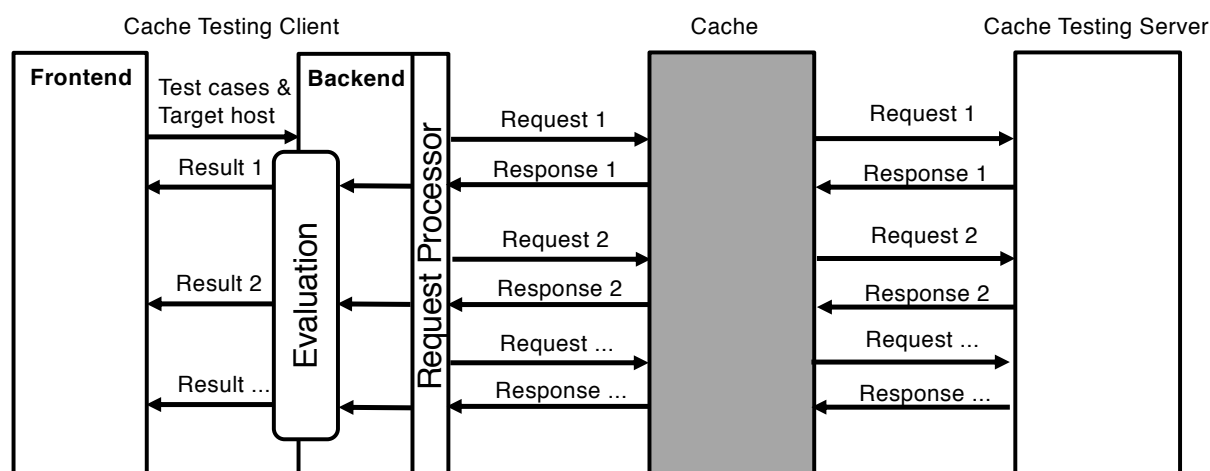


Figure 5.2: Architecture of universal shared cache testing system based on [NLF18]

The cache testing client is a web application which consists of a frontend where the test vectors can be defined and a backend which executes the requests as well as evaluates the returning responses. The frontend provides a user interface which requires two mandatory input arguments: the target host and a set of test cases defined by the provided test case specification language. The request processor reads the supplied test cases, converts each test case into a series of requests and schedules them for delivery to the server. On their way to the server, the requests pass the cache. In order to unambiguously distinguish whether a response is originated by the cache testing server or supplied by the web caching system, the request processor includes specific meta information to the requests which will be replayed by the cache testing server. By means of the replayed meta information included in the responses, our cache testing client is able to infer several properties including whether the content is a cached copy or if other expected testing requirements are met.

### 5.5.2 Test Case Suite

The developed testing tool for shared web caches comes with a test case suite of 397 tests that can be used out of the box. This suite originates from [NLF18] and tests the conformance with various HTTP and web caching standards. All 397 tests are defined using a custom specification language. By this means, the base test suite can be easily explored and even extended with individual tests. Listing 5.1 shows an example test case which is defined with the proposed test case specification language.

```

1 ## Testing freshness lifetime of 60 seconds
2 GET /rsc -c 'Accept:application/json;Cache-Control:no-store'
3     -s 'Cache-Control:max-age=60'
4     -p 30
5     -e 'ch:false;st:200'
6 GET /rsc -c 'Accept:application/json'
7     -e 'ch:true;st:200'
8     -p 35
9 GET /rsc -c 'Accept:application/json'
10    -e 'ch:false;st:200'

```

Listing 5.1: Example test case defined by the provided specification language

The provided test case specification language is based on the command line tool curl [Ste18]. Each test case starts by a descriptive title followed by one or multiple commands (see Listing 5.1). A command contains two mandatory and several optional arguments. The first mandatory argument is the request method, i.e. GET, POST, PUT, PATCH and DELETE. The second mandatory argument represents the request target which is the URL omitting the scheme and host parts. Listing 5.1 contains three commands: lines 2 to 5 define the first command and the second one is specified from lines 6 to 8. Lines 9 to 10 cover the third and last command. Each command specifies a request and its expected response. The first command demands to generate a GET request targeting the URL `/rsc`. Optionally, the test case specification language allows to add one or more headers to the request. This is done by the `-c` parameter flag. Multiple headers must be separated by semicolon. In the example, the first command requires to add two headers to the request: (i) the `Accept` header with the value `application/json` and (ii) the `Cache-Control` header containing the value `no-store`. The `-s` parameter flag instructs the cache testing server to add the specified headers to the resulting response. The `-p` parameter flag forces the client to wait a certain amount of seconds before executing the next request. Such breaks between requests are, e.g., required for analyzing whether a cache complies with the freshness lifetime requirements. With the `-e` parameter flag, users can define the to be expected properties of the response. These can be either RFC 7234 requirements or expectations of the own caching policy. The first expected property in the first command `ch:false` (see line 5) assumes that the resulting response originates from the server and is not a cached copy. The second expected property `st:200` demands the response to contain the status code 200 Ok.

Figure 5.3 shows the request/response flow executed by the web cache testing tool when executing the test case specified in Listing 5.1. Accordingly, the first request includes the GET method targeting the URL path `/rsc`. Note that the GET request contains the `Accept` and `Cache-Control` headers signalling the retrieval of an `application/json` resource representation that must not be a stored copy from a cache. The request also includes the `X-Response` header which contains the headers that the cache testing server has to add to the resulting response. Therefore, the resulting response includes the `Cache-Control` header with the value `max-age=60` which allows all caches to store the content for 1 minute.

All requests created by a our cache testing client comprise the `X-Id` header with a unique random generated id. This token is returned by the cache testing server in case the request reaches it. The cache testing server adds the id in the `X-Id` header of the response as well as the body. The generated id including in the request and its resulting response allows evaluating whether a received response has been replayed by a caching system or originates from our cache testing server. For instance, by means of the id inside the `X-Id` header, our tool can assess that the first response is sourced from the server as it contains the same id as the corresponding

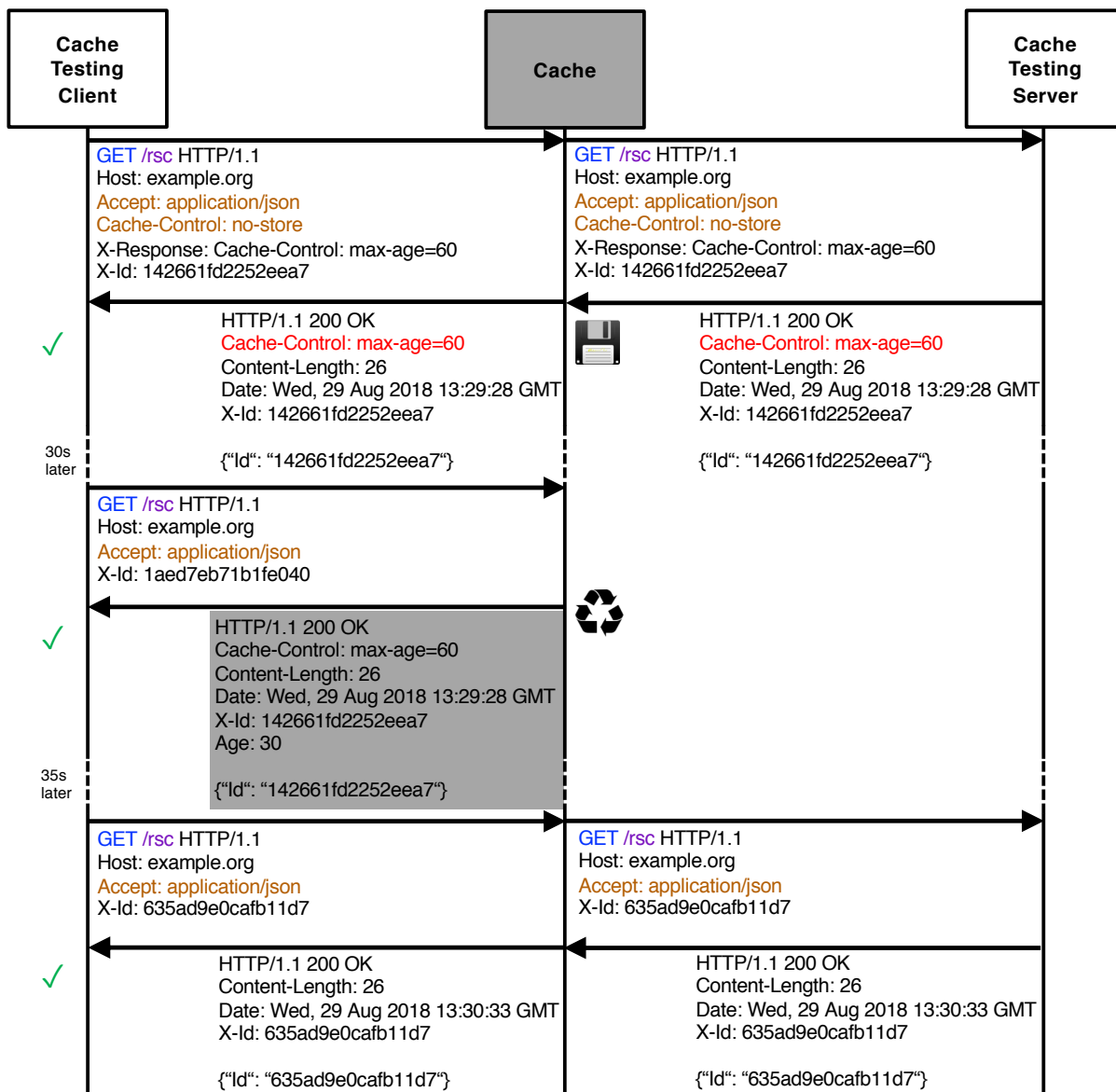


Figure 5.3: Request/Response flow produced by the web cache testing tool according to the test case specified in Listing 5.1

request. The second response instead is a recycled copy from a shared cache (emphasized by the gray background), since it includes the id of the first request and not the one from the second request. Another indicator for a cached response is the `Age` header. Moreover, the cache testing server also appends the `Date` header which represents the response creation time. This date value is also another information, which can be used to assess whether the response is reused or new.

Figure 5.3 illustrates that all the three responses comply with the expected properties specified in the test case specification (see Listing 5.1). The first response is a new one from the server and contains the status code `200 Ok`. The second is a recycled response from the web caching systems comprising the status code `200 Ok` as well. The final response is a new one forwarded by the cache from the server as the previous reused response is expired.

The introduced web cache testing tool with the base test suite is provided as public domain software can be obtained from <https://github.com/DASCologne/Cache-Testing-Tool>.



## 5.6 Empirical Study Results

To evaluate the introduced shared web cache testing system and to explore available shared caching systems we analyzed popular widely deployed backbone and client-side as well as server-side caching solutions. For the client- and server-side, we choose the proxy caches Apache HTTP Server v2.4.18 (Apache HTTPD)<sup>3</sup>, Nginx v1.10.3<sup>4</sup>, Varnish v5.1.1<sup>5</sup>, Apache Traffic Server v7.1.1 (Apache TS)<sup>6</sup>, Wingate v9<sup>7</sup> and Squid v3.5.12<sup>8</sup>. Additionally, we analyze the Amazon CloudFront CDN<sup>9</sup>.

All tested caching systems were left in their default configuration, with some small exceptions. All proxy caches except Wingate enable the option to add the X-Cache-\* headers on each response. This header provides additional details on the cache status of the respective response, e.g., whether a cached copy of it has been reused. This meta information is a complementary indicator for studying the caching behavior. We enabled this feature to have an additional source for validating our implementation.

Table 5.1 illustrates and this section discusses the most significant results of our empirical analysis on caching systems. The full evaluation can be accessed at <https://cachetester.github.io/cachetest>.

	Apache HTTPD	Nginx	Varnish	Apache TS	Wingate	Squid	Amazon CDN
<b>Explicit Freshness Lifetime</b>							
max-age	●	●	○	●	●	●	●
s-maxage	●	●	○	●	●	●	●
s-maxage & max-age	●	●	○	●	●	●	●
Expires	●	●	○	●	○	●	●
max-age & Expires	●	●	○	●	⦿	●	●
s-maxage & Expires	●	●	○	●	⦿	●	●
s-maxage, max-age & Expires	●	●	○	●	⦿	●	●
<b>Implicit Freshness Lifetime</b>							
Last-Modified	⦿	○	⦿	○	●	○	○
Date	○	○	⦿	○	○	○	●
<b>Explicit Freshness Validation</b>							
ETag & no-cache	○	○	○	○	⦿	○	⦿
Last-Modified & no-cache	○	○	○	○	⦿	●	⦿
Date & no-cache	○	○	○	○	○	○	⦿
<b>Implicit Freshness Validation</b>							
ETag	○	○	○	○	⦿	○	○
Last-Modified	○	○	○	○	⦿	●	○
Date	○	○	○	○	○	○	○
<b>Invalidation</b>							
PUT	●	○	○	●	○	●	○
DELETE	●	○	○	●	○	●	○
POST	●	○	○	●	○	●	○
PATCH	○	○	○	○	○	●	○
<b>Client-originated Policies</b>							
max-age	● △	○	○	○	● △	● △	○
max-stale	●	○	○	○	●	○	○
no-store	● △	○	○	○	○	○	○
min-fresh	●	○	○	●	●	○	○
only-if-cached	●	○	○	○	○	●	●
no-cache	○	○	○	○	○	○	○
<b>Security</b>							
Prefer explicit freshness lifetime	●	●	●	○ △	●	●	○ △
Prohibiting two Content-Length header	●	●	●	○ △	●	●	●
Storing response requiring AuthN	○	● △	○	○	○	○	○
Storing Set-Cookie header	● △	○	○	●	●	●	●

Legend: ● RFC 7432 compliant, ○ none RFC 7432 compliant, ⦿ partially RFC 7432 compliant, △ potentially vulnerable

Table 5.1: Results of the empirical analysis of caches obtained by the automated evaluation of 397 test cases

3. <https://httpd.apache.org>
4. <http://nginx.org>
5. <https://varnish-cache.org>
6. <http://trafficserver.apache.org>
7. <https://www.wingate.com>
8. <http://www.squid-cache.org>
9. <https://aws.amazon.com/cloudfront>



## 5.6.1 Freshness Lifetime

### Proxy Caches

Our analysis shows that Varnish and Wingate do not comply or only partially comply with the explicit freshness lifetime policies of RFC 7234. Varnish ignores almost all freshness lifetime definitions with `max-age`, `s-maxage` or the `Expires` header and stores each response, regardless whether a freshness lifetime is defined or not, for a fixed predefined time span. By default this is 120 seconds. This parameter represents the only option for adjusting the freshness lifetime. The only way to influence the cache behavior is to include `max-age=0`, `no-store`, `no-cache` and `private` in the response. Then Varnish does not cache or reuse the corresponding response and forwards recurring requests to the origin server. Wingate considers the values and the order of the `max-age` and `s-maxage` directives, but does not support the `Expires` header. If no explicit freshness lifetime definitions are set, but the `Last-Modified` header and a suitable status code such as `200 Ok` are present, Apache HTTPD and Wingate store responses and calculate the implicit freshness lifetime by multiplying the date value of the `Last-Modified` header with a predefined factor.

All proxies provide a configuration options for stating self-defined implicit caching policies. These requirements can be, e.g., a distinct data format, URL path or file suffix. As described in Section 5.2, if implicit caching is enabled, caches are still required to take account of the explicit requirements of the origin server and must prefer the server-side caching prerequisite over the implicit ones. Otherwise the disobey of explicit caching, which is shown by Varnish, may induces misbehavior in the business logic as responses are stored and reused longer as indicated. Not prioritizing the explicit caching declaration over the implicit requirements is one prerequisite for the web deception attack [Gil17]. With the help of our tool, we are able to detect that web application using Apache TS might be vulnerable to this threat. If implicit caching preconditions are specified via the configuration, the proxy ignores any explicit caching definitions set by the origin server and stores responses according to the implicit caching policy only. That is, a sensitive response which fulfills the precondition, but contains the `Cache-Control` header including `no-store` or `max-age=0`, is still stored. In conjunction with a flaw in the routing process, a web deception attack can exploit this malfunction to retrieve sensitive information.

### CDNs

The Amazon CloudFront CDN specifies an implicit freshness lifetime of 24 hours for responses without the `Expires` or the `Cache-Control` header, if a suitable status code such as `200 Ok` is given. Amazon does not utilize the `Last-Modified` header to derive an implicit freshness lifetime, as it uses this header to initiate a validation request. The implicit freshness lifetime of 24 hours is also overruled, if the origin server sets an explicit freshness lifetime. The CDN honors all freshness lifetime definitions with `max-age`, `s-maxage` and `Expires`. As Amazon complies with all explicit and implicit freshness lifetime requirements of RFC 7234, this proper caching behavior has no negative consequences in terms of performance, security and privacy.

Optionally, Amazon also allows setting a self-defined implicit freshness lifetime for distinct resources. However if this configuration option is set, the CDN might be vulnerable to the web

deception attack. Here, each content is stored with the predefined freshness lifetime even if the response contains `no-store` or `max-age=0`.

## 5.6.2 Freshness Validation

### Proxy Caches

None of the proxy caches comply with all explicit freshness validation control directives. Wingate and Squid are the only proxies, which partially complies with some requirements. If the response contains the `Last-Modified` header, Wingate uses its value to perform a conditional request but does not update the header of the returned cached response when the validation is successful. However, Wingate shows non-compliance and odd behavior when the validation request with an opaque token from the `ETag` header is successful. Here, Wingate forwards the response of the origin server containing `304 Not Modified` status code without any body. For a client such a response is useless, as it does not content any meaningful information. Squid performs an explicit validation request if a response contains the `Last-Modified` and `Cache-Control` headers with `no-cache`. Responses with `no-cache` and an opaque token are not validated by an conditional request, though. Here, it issues a request without any conditions always fetching a fresh response containing a full body instead. Moreover, Squid does not perform a implicit conditional requests for responses containing the `ETag` header. Wingate executes a implicit validation requests for responses which include the `ETag` or `Last-Modified` header. As with responses with a `no-cache` control directive and the `ETag` header, Wingate forwards the server's response with a `304 Not Modified` status code to the client if the resource has not changed in the meantime. Apache, Nginx, Varnish and Apache TS never perform a freshness validation for recurring requests. They always issue a request without any conditions retrieving the full response. Moreover, all analyzed proxy caches never use the time-variant parameter of the `Date` header to perform an explicit or implicit validation request.

The freshness validation with conditional requests instead of retrieving the same full response with body on each request saves a lot of workload on the server as well as network traffic when used properly. Therefore, caches that ignore conditional requests indicators and requirements impair the performance of web applications.

### CDNs

Explicit freshness validation is supported for responses containing the `ETag` or `Last-Modified` header. That is, the CDN initiates a conditional request, if the origin server forces the cache to do so by declaring `no-cache`. However, Amazon does not update the headers, it reuses the header of the cached response. A validation is not performed if the `ETag` and `Last-Modified` header are missing. Both headers are not used to perform an implicit validation. Responses with these headers are stored implicitly for 24 hours, if the origin server does not specify any freshness lifetime.

Unlike some analyzed proxy caches, Amazon does perform an explicit validation for responses with certain validation tokens. This reduces data traffic and improves performance. However, not updating the header in case of a successful validation is a violation of the standard and may affect the business logic.

### 5.6.3 Client-originated Caching Policies

#### Proxy Caches

Apache HTTPD is the only caching system which honors all client-side control directives except `no-cache`. The other proxies only support the control directives partially.

Not respecting the explicit caching requirement of the client may induce the same consequences as not considering the server-side ones. On the one hand, refusing the client's caching requirements in some cases is crucial for retaining the scalability. According to RFC 7234, the client can use the `no-store` or `max-age=0` directive to force the cache to fetch each response from the origin server without considering cached copies. These requests may reduce the scalability and increase the workload of the origin server as each request is forwarded to the endpoint. Likewise, bypassing a cache with these control directives enables the option to execute a DDoS attacks similar to the vulnerability described by Triukose et al. [TAR09]. To avoid such threats, a cache can simply ignore and disobey cache-related client headers. On the other hand, disrespecting client-side control directives which require to fetch a response from the origin server can lead to users becoming victims of cache poisoning attacks. Client intending to circumvent this attack must add `no-store` or `max-age=0` to the `Cache-Control` header which mandate the cache to fetch a new response from the origin server. Such circumvention is only possible, if web caching systems support and honor the control directives of the client.

#### CDNs

Amazon respects the `only-if-cached` control directive only. As with the proxy caches, not honoring client-side control directives hinders clients from bypassing a cache. This retains the performance in some cases. However, ignoring the client caching requirements removes the option for service consumers to circumvent cache poisoning attacks.

### 5.6.4 Invalidation of Freshness

#### Proxy Caches

Our analysis reveals that only Apache TS, Apache HTTPD and Squid perform an invalidation of a stored response if a response to a request containing an unsafe HTTP method is successful. Moreover, Squid even triggers an invalidation if requests include the `PATCH` [DS10] method is successfully processed. Apache HTTP and Apache TS does not consider `PATCH` as valid unsafe method and still return a stored response after a successful `PATCH` request. Nginx, Varnish and Wingate do not perform an invalidation in case of a successful `PUT`, `DELETE`, `POST` or `PATCH` request. Instead of fetching a new response from the origin they return a cached copy if the freshness lifetime has not expired.

Ignoring unsafe methods as indicator for invalidating a response's freshness leads to misbehavior in the business logic. Resources that have been removed or changed are not updated by the cache inducing that users obtain stale responses. Especially, providers of REST-based web services [RR08] might be affected by this malfunction.

## CDNs

Amazon does never perform an invalidation after a successful request with an unsafe method. It always returns a stored response even though the target resource is changed or deleted. This improper behavior may lead to severe consequences for clients using REST-based web services deploying AWS CloudFront as CDN.

### 5.6.5 Security

#### Proxy Caches

In web applications, the `Authorization` header is utilized for transferring credentials intended for authentication. To avoid middleboxes from storing responses requiring authentication, RFC 7234 prohibits a shared cache to store responses resulting from requests containing the `Authorization` header. The standard allows to cache protected responses only if the content provider explicitly permits it by specifying a freshness lifetime with the `max-age/s-maxage` directive or the `Expires` header.

Apache HTTPD, Varnish, Apache TS and Wingate do not store and reuse any response which results from a request containing the `Authorization` header. They even ignore all explicit freshness lifetime headers and control directives. Nginx and Squid do cache and recycle responses for equivalent requests, if the origin server specifies an explicit freshness lifetime. This compliant behavior provides a performance enhancement for resources requiring an authentication as the cache omits the verification process for the origin server. However, Nginx and Squid also return a cached response to a request embodying the `Authorization` header, if an equivalent request does not contain the `Authorization` header. That is, any client can retrieve a stored response requiring an authentication via these two caches without knowing the valid credentials. Unfortunately, such a critical behavior is in conformance with RFC 7234, since the standard does not require a cache to verify the `Authorization` header credentials before reusing the response. To prevent disclosure of access protected data residing in a shared cache, the origin server must declare the `Authorization` header as an additional cache key. Then, a cache must compare the value within the `Authorization` header for every equivalent request. Only if the value in the `Authorization` of the recurring request matches with the stored one, the cache will return the response. Still, a system engineer needs to be aware of this fact in order not to deploy caching components leaking sensitive data.

The value inside the `Set-Cookie` header is also commonly utilized for authentication purposes. This header represents a vital meta information for transferring a session id to the client. With this session id included in the `Cookie` request header, users can be identified and authenticated by web applications. Hence, the disclosure of a `Set-Cookie` header to an attacker induces severe consequences. We found that Apache HTTPD stores and reuses responses with the `Set-Cookie` header when an explicit freshness lifetime is set. The critical issue of this behavior is that the proxy does not remove the `Set-Cookie` header from the response when the content is reused. Hence, session ids inside `Set-Cookie` can be accidentally passed to other users or an attacker being aware of this issue may try to steal cookies. The problem with this behavior is that the storage of responses comprising the `Set-Cookie` header by shared caches is not prohibited according to RFC 7234. The HTTP standard only mentions that `Set-Cookie` is not a header which prevents caching. Origin servers intending to store

such responses should consider this aspect and are encouraged to use appropriate headers. This means that Apache HTTPD conforms to RFC 7234 in this regards. But this compliant behavior can lead to the disclosure of security-related data. The other investigated proxies do not show this behavior. Nginx and Varnish do not store any response containing the `Set-Cookie` header even if the response includes an explicit freshness definition. Apache TS, Wingate and Squid store such a response, but remove the `Set-Cookie` header from the response if they reuse it for recurring requests. This an ideal behavior for balancing security and scalability as it prevents attackers from stealing cookies while considering the cacheability of responses containing publicly accessible content. Still, the proper handling of the `Set-Cookie` header is mentioned briefly in a side-note by the RFC 7234 making it error prone for unaware developers. To simulate the behavior of Apache TS, Wingate and Squid, service providers can utilize the `private` directive. As described in Section 5.2, filling the `private` directive with header names (e.g `private="Set-Cookie"`) allows a shared cache to store responses containing the `Set-Cookie` header but with the exception that the intermediary must remove the `Set-Cookie` header when storing it. Our analysis found that only Apache HTTPD supports the `private` directive containing headers. The other proxy caches do not store any response containing a `private` directive regardless whether this parameter includes headers or not.

Moreover, we found that Apache TS permits to send a request with two `Content-Length` headers. This is a critical behavior, which can lead to a request smuggling attack. We will investigate this issue in further work to analyze if this issue can cause such an attack.

Another noteworthy finding, which may cause security issues, is that all analyzed caching systems incorporate the whole URL including the query part as cache keys by default. This means adding a random string to the query allows bypassing the cache. Considering the fact that some proxies (e.g. Squid, Varnish and Apache TS) can be used to build a CDN, this compliant behavior can be exploited to perform DDoS attacks such as the one demonstrated by Triukose et al. [TAR09]. All analyzed proxy caches provide the option to disable the query part or certain query parameter as cache keys via configuration. However, server providers using these caching systems must be aware of the default behavior and must adjust the configuration according to their conditions. These settings must be tested carefully with the aim that the cache is able to understand the application-specific definitions of equivalent requests.

## CDNs

By default, Amazon removes the `Authorization` header from the request and forwards the request without this header to the origin server. The resulting response is stored implicitly or explicitly if freshness lifetime definitions are present. If content providers want that Amazon forwards the `Authorization` header to the origin server, then they need to add it to a request header whitelist. This whitelist can be considered as the request header cache keys. If this is done, Amazon does not remove this header from the request anymore and forward the request with the `Authorization` header to the origin server. The resulting responses are stored implicitly or explicitly according to the given policy. Moreover, the CDN compares the `Authorization` header for recurring requests and returns stored content only if the credentials match with the values of the cached response. Such a handling of the `Authorization` header hinders attackers from obtaining protected responses without the knowledge of the valid credentials.

Responses with the `Set-Cookie` header are stored implicitly or explicitly depending on the header information. However, the CDN removes the `Set-Cookie` header from the response even when this resource is requested for the first time. Removing the `Set-Cookie` header in each response hinders the transfer of the session id to the clients. This behavior can hamper authentication procedures of web applications, which use the AWS CloudFront CDN.

With the default settings, Amazon cannot be exploited to perform DDoS attacks as introduced by Triukose et al. [TAR09], when using exact the same technique. Amazon ignores the query part as a cache key member by default. If web services still require incorporating query parameters for delivering different responses, they can either include the query parameter in the settings or they can use a whitelist. In this whitelist, service providers can define distinct query parameters, which are processed by the origin server to return different content. If query parameters are included in the list, Amazon only stores responses based on the defined parameters and the corresponding parameters values. All other parameters (e.g. an appended random string) and associated values are ignored. However, the whitelist still enables clients to penetrate through an edge server cache if they include a random string as value in an allowed query parameter. For instance, if the query parameter `a` is in the whitelist, URLs with same query parameter but a different corresponding parameter value (e.g. `http://example.org?a=<randomString>`) always provoke the CDN to forward the request to the origin server. For this paper, we do not investigate if this setting might be exploited to conduct DDoS attacks.

## 5.7 Conclusion and Outlook

The obtained and discussed results emphasize that the analyzed web caching systems contain many malfunctions. In summary, we found that these malfunctions in caches result from (i) the non-conformance with RFC 7234, (ii) contradictions in RFC 7234, (iii) misconfigurations and (iv) vulnerabilities in the business logic of web caching systems.

The issues and consequences of malfunctioning caches introduced in this paper show that proper web caching is not a trivial task. Web caching systems are required to consider issues that are beyond the definitions in the standard in order to provide a dependable caching systems promoting performance and protection. There is not a reliable caching policy, which can be applied for all caching systems. A proper caching guideline needs to be adjusted and tested according to the requirements of the corresponding web application. Standardization needs to be improved in order to reduce the current lock-in to specific caching solutions. This all emphasizes the need for meaningful tools and methodologies for investigating conformance and security issues of web caching systems before combining them with web applications in production.

Further work will investigate caching facets in respect to security services. The aim is to propose robust protection means, which do not impair the scalability and performance of ULS systems.

# Chapter 6

## Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack

### Summary of this publication

**Citation** H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack*. In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019. URL: <https://doi.org/10.1145/3319535.3354215>

**Status of Paper** Published

**Type of Paper** Research Paper (Conference)

**Ranking** GGS: A++, CORE: A++, LiveSHINE: A++, Microsoft Academics: A++

**Aim** In this paper, we introduce and analyze the Cache-Poisoned Denial of Service (CPDoS) attack, a new class of web cache poisoning vulnerability.

**Methodology** We conduct multiple extensive empirical studies to evaluate the practicability CPDoS in real world websites. We study the caching behavior of error pages, the handling of erroneous request headers and, the identification of potentially vulnerable systems.

**Contribution** With these experiments, we show the practical relevance by identifying one proxy cache product and five CDN services that are vulnerable to CPDoS. Amongst them are prominent solutions that in turn cache high-value websites. The consequences are severe as one simple request is sufficient to paralyze a victim website within a large geographical region.

**Co-authors' contribution** See Paper 5 in Section 1.1.1.



## 6.1 Introduction

Contemporary distributed software systems require to scale at large in order to efficiently handle the sheer magnitude of requests stemming, e.g., from human users all over the globe or sensors scattered around in an environment. A common architectural approach to cope with this requirement is to design the system in layers composed of distinct intermediaries. Application-level messages travel through such intermediate systems on their path between a client and a server. Common intermediaries include caches, firewalls, load balancers, document routers and filters.

The caching of frequently used resources reduces network traffic and optimizes application performance and is one major pillar of success of the web. Caches store recyclable responses with the aim to reuse them for recurring client requests. The origin server usually rules whether a resource is cacheable and under which conditions it can be provided by a caching intermediate. Cached resources are unambiguously identified by the cache key that consists most commonly of the HTTP method and the URL, both contained in the request. In case a fresh copy of a requested resource is contained in an intermediate cache, the client receives the cached copy directly from the cache. By this, web caching systems can contribute to an increased availability as they can serve client requests even when the origin server is offline. Moreover, distributed caching systems such as Content Distribution Networks (CDNs) can provide additional safeguards against Distributed DoS (DDoS) attacks.

A general problem in layered systems is the different interpretation when operating on the same message in sequence. As we will discuss in detail in Section 6.3, this is the root cause for attacks belonging to the family of "semantic gap" attacks [JS12]. These attacks exploit the difference in interpreting an object by two or more entities. In the context of this paper the problem arises when an attacker can generate an HTTP request for a cacheable resource where the request contains inaccurate fields that are ignored by the caching system but raise an error while processed by the origin server. In such a setting, the intermediate cache will receive an error page from the origin server instead of the requested resource. In other words, the cache can get poisoned with the server-generated error page and instrumented to serve this useless content instead of the intended one, rendering the victim service unavailable. This is why we denoted this novel class of attacks "Cache-Poisoned Denial-of-Service (CPDoS)".

We conduct an in-depth study to understand how inconsistent interpretation of HTTP requests in caching systems and origin servers can manifest in CPDoS. We analyze the caching behavior of error pages of fifteen web caching solutions and contrast them to the HTTP specifications [FR14c]. We identify one proxy cache product and five CDN services that are vulnerable to CPDoS. We find that such semantic inconsistency can lead to severe security consequences as one simple request is sufficient to paralyze a victim website within a large geographical region requiring only very basic attacker capabilities. Finally, we show that the CPDoS attack raises the paradox situation in which caching services proclaim an increased availability and proper defense against DoS attacks while they can be exploited to affect both qualities.

Overall, we make three main contributions:

1. We present a class of new attacks, "Cache-Poisoned Denial-of-Service (CPDoS)", that threaten the availability of the web. We systematically study the cases in which error pages are generated by origin servers and then stored and distributed by caching systems. We introduce three concrete attack variations that are caused by the inconsistent treatment

of the `X-HTTP-Method-Override` header, header size limits and the parsing of meta characters.

2. We empirically study the behavior of fifteen available web caching solutions in their handling of HTTP requests containing inaccurate fields and caching of resulting error pages. We find one proxy cache product and five CDN services that are vulnerable to CPDoS. We have disclosed our findings to the affected solution vendors and have reported them to CERT/CC.
3. We discuss possible CPDoS countermeasures ranging from cache-ignoring instant protections to cache-adhering safeguards.

## 6.2 Foundations

The web is considered as the world's largest distributed system. With the continuous growing amount of data traveling around the web, caching systems become an important pillar for the scalability of the web [BO00b]. Web caching systems can occur in various in-path locations between client and origin server (see Figure 6.1). Another distinction point is the classification in private and shared caches. Private caches are only allowed to store and reuse content for one particular user. Client-internal caches of web browsers are one typical example of private cache as they store responses for a dedicated user only. On the other hand, client-side and server-side caches—also known as proxy caches—as well as CDNs deployed in the backbone of the web belong to the family of shared caches, since they provide content for multiple clients. Some web applications may also include a server-internal cache. These caching systems usually support both access policies, i.e., they are able to serve cached resources to multiple users or to one client exclusively.

The cache policy is governed by the content provider by specifying caching declarations defined in RFC 7234 [FNR14]. The web caching standard defines a set of control directives for instructing caches how to store and reuse recyclable responses. The `max-age` and `s-maxage` attributes in the `Cache-Control` response header define, e.g., the maximum duration in seconds that the targeted content is allowed to reside in a cache. The keyword `max-age` is applicable to private and shared caches whereas `s-maxage` only applies to shared web caching systems. Content providers can also use the `Expires` header with an absolute date to define a freshness lifetime. As with `max-age`, the `Expires` is adoptable for private and shared caches. A stored response in a cache is considered as fresh, if it does not exceed the freshness lifetime specified by `max-age`, `s-maxage` and the `Expires` header. If a content provider wishes to permit a certain content to be saved by private caches only, it adds the `private` directive to the `Cache-Control` header. Content providers which do not want that a certain response is stored and reused by any cache have to include the keyword `no-store` in the `Cache-Control` header. The control directives `must-revalidate`, `proxy-revalidate` and `no-cache` in the `Cache-Control` header instruct how to verify the freshness of a response, in case a content is expired or no freshness lifetime information is available. All mentioned control directives enable a content provider to define caching policies in an explicit manner.

If no explicit caching directive is present in a response, a web caching system may store and reuse responses implicitly when certain conditions are met. One requirement which permits caches to store content implicitly is a response to a GET request. Responses to unsafe methods including POST, DELETE and PUT are not allowed to be cached. Moreover, responses to

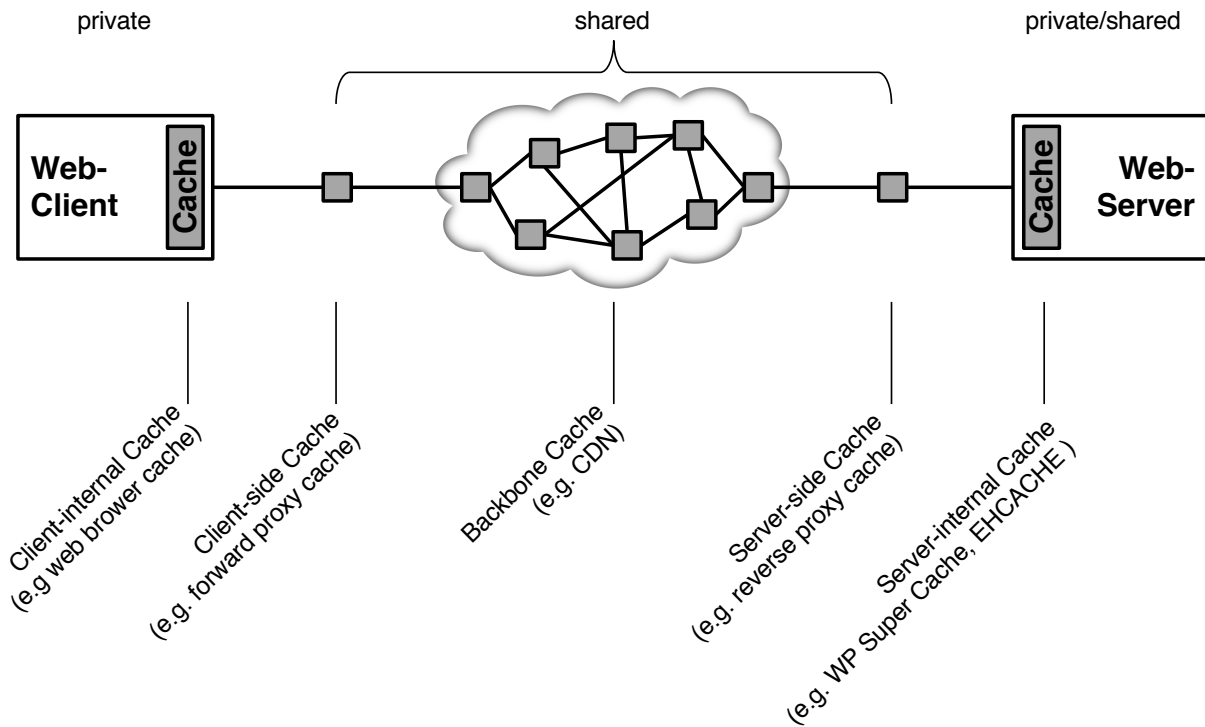


Figure 6.1: Different types of web caching systems classified by location and resource access policy [NLF19a]

GET method must contain defined status codes including, e.g., 200 Ok, 204 No Content and 301 Moved Permanently. Here, caches are allowed to derive a freshness lifetime by using heuristics. Many web applications instruct web caching systems to define an implicit freshness lifetime for images, scripts and stylesheets as these file types are considered as static content. Static content refers to data which does not change frequently. Therefore, storing and reusing such resources is considered as best practice for optimizing the performance.

In some cases, it is also very useful for content providers to cache certain error messages. For instance, the status code 404 Not Found, which indicates that the origin server does not have a suitable representation for the requested resource, is permitted to be cached implicitly. The 405 Method Not Allowed declaring the request action is not supported for the targeted resource can be cached implicitly as well.

### 6.3 Security Threats in Web Caching Systems

Using web caching systems provides many advantages in terms of optimizing communication and application performance. However, much work has shown that web caches can also be exploited to affect the privacy and reliability of applications. Web cache poisoning attacks, e.g., are a serious threat that has been emerging over the past years. Amongst them is the *request smuggling* [Lin+05] attack which occurs when the web caching system and the origin server do not strictly conform to the policies specified by RFC 7234. In this particular attack, the attacker can send a request with two Content-Length headers to impair a shared cache. Even though the presence of two Content-Length headers is forbidden as per RFC 7234, some HTTP engines in caches and origin servers still parse the request. Due to the duplicate headers, the

malformed request is able to confuse the origin server and the cache so that a harmful crafted response can be injected to the web caching system. This malicious response is then reused for recurring requests.

The *host of troubles* [Che+16] attack is another vulnerability targeting shared caches. As with the previous attack, it exploits a violation of the web caching standard that gets interpreted differently by the involved system layers. Here, the attacker constructs a request with two `Host` headers. These duplicate headers induce a similar misbehavior in the cache and origin server as the request smuggling attack. Likewise, a malicious response is injected to poison the cache.

Another attack that targets to poison web caches is the *response splitting* [Kle04] attack. Unlike the two aforementioned vulnerabilities, where a flaw in the shared cache itself is one reason why the attack is successful, the response splitting attack exploits a parsing issue in the origin server only. Here, an attacker utilizes the fact that the HTTP engine of the origin server does not escape or block line breaks when replaying a request header value in the corresponding response header. A malicious client can exploit this by dividing the response in two responses. The aim of this attack is to poison the intermediate cache with the malicious content contained in the second response.

James Kettle [Ket18c] presented a set of cache poisoning attacks which result from a misbehavior in web application frameworks and content management systems respectively. With the introduced techniques, James Kettle was able to compromise shared web caching systems of well-known companies.

All introduced attacks aim at poisoning shared caches with malicious content that gets served by the victim caches for recurring requests of benign clients. Private caches such as the web browser cache are not affected by the mentioned attacks. However, browser caches are not immune to this class of attacks. Jia et al. [Jia+15] present browser cache poisoning (BCP) attacks. In their study they find that many desktop web browsers are susceptible to BCP attacks.

The *web cache deception* [Gil17] attack targets to poison a shared cache with sensitive content. Here, the attacker exploits a RFC 7234 violation of a shared cache which still stores responses even though it is prohibited. In combination with an issue in the request routing of the origin server, the author was able to retrieve account information of third parties out of the cache.

Triukose et al. [TAR09] showed another attack vector that utilizes web caching systems to paralyze a web application. Unlike the presented threats, this attack does not intend to poison a cache with harmful content or to steal sensitive data. The goal of Triukose et al. was to provoke a DoS attack with the aid of a mounted CDN. The authors utilized the infrastructure of a CDN, which comprises of many collaborating edge cache servers. With the use of a random string appended to the URL query, Triukose et al. were able to bypass any edge cache servers so that the CDN forwards every request to the origin server. To create a DoS attack, the authors send multiple requests with different random query strings to all edge cache servers within the CDN. As the edge cache servers forward all of these requests to the origin server, the huge amount of requests reaching the origin server generates a high workload with the consequence that the web application cannot process any further legitimate request.

The root cause of almost all of the presented attacks lies in the different interpretation of HTTP messages by two or more distinct message processing entities, which is known as the *semantic gap* [JS12]. Vulnerabilities stemming from the semantic gap are manifold [Kle04; Che+16; Som+11]. In relation to web caches the request smuggling, host of troubles and response

splitting attacks exploit this gap between a cache and an origin server. Here, a discrepancy in parsing duplicate headers or line breaks leads to cache poisoning.

In the next section we introduce a new class of attacks against web caches, the Cache-Poisoned Denial-of-Service (CPDoS) attack. It exploits the semantic gap between a shared cache and a origin server for poisoning the cache with error pages. As a consequence, the cache distributes error pages instead of the legitimate content after being poisoned. Users perceive this as unavailable resources or services. In contrast to the DDoS attack introduced by Triukose et al., CPDoS require only very basic attack skills and resources.

## 6.4 Poisoning Web Caches with Error Pages

The general attack idea is to exploit the semantic gap in two distinct HTTP engines—one contained in a shared cache and the other in an origin server. More specifically, the baseline of the newly introduced variant of web cache poisoning takes advantage of the circumstance that the deployed caching system is more lax or focused in processing requests than the origin server (see Figure 6.2). An attacker can make use of this discrepancy by including a customized malicious header or multiple harmful headers in the request. Such headers are usually forwarded without any changes to the origin server. As a consequence, the attacker crafted request runs through the cache without any issue, while the server-side processing results in an error. Henceforth, the server’s response is a respective error, which will be stored and reused by the cache for recurring requests. Each benign client making a subsequent GET request to the infected URL will receive a stored error message instead of the genuine resource form the cache.

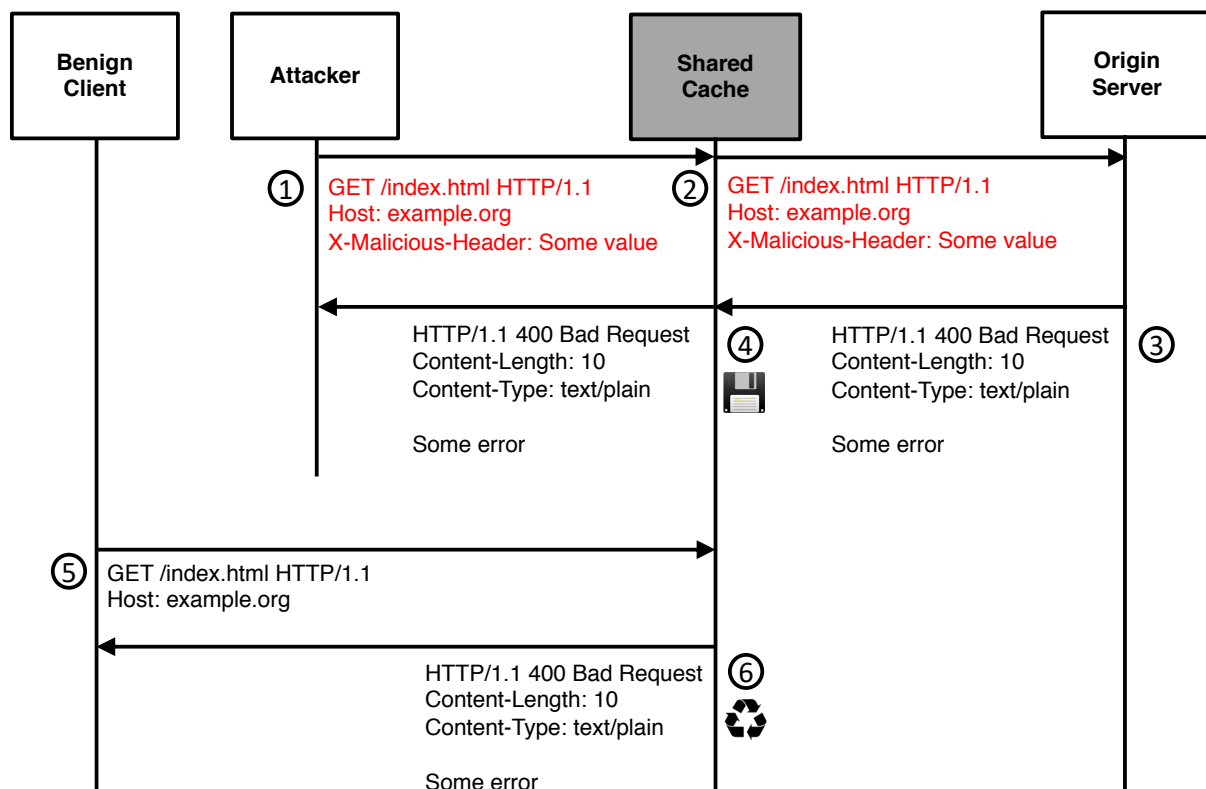


Figure 6.2: General construction of the Cache-Poisoned Denial-of-Service (CPDoS) attack

It is worth noting that one simple request is sufficient to replace the genuine content in the cache by an error page. This means that such a request remains below the detection threshold of web application firewalls (WAFs) and DDoS protection means in particular, as they scan for large amounts of irregular network traffic.

The consequences for the web application depend on the content being illegitimately replaced with error pages. It will always affect the service's availability—either parts of it or entirely. The most harmless CPDoS renders images or style resources unavailable. This influences the visual appearance of parts of the application. In terms of functionality it is still working, however. More serious attacks targeting the start page or vital script resources can render the entire web application inaccessible instead. Moreover, CPDoS can be exploited to block, e.g., patches or firmware updates distributed via caches, preventing vulnerabilities in devices and software from being fixed. Attackers can also disable important security alerts or messages on mission-critical websites such as online banking or official governmental websites. Imagine, e.g., a situation in which a CPDoS attack prevents alerts about phishing emails or natural catastrophes from being displayed to the respective user.

When considering the low efforts for attackers, the high probability of success, the low chance of being detected and the relatively high consequences of a DoS then the introduced CPDoS attack poses a high risk. Hence, it is worthwhile investigating under which conditions CPDoS attacks can occur in the wild. For this reasons we first compiled a complete overview on cacheable error codes as specified in relevant RFCs [HM98], [Mas98], [NL00], [Dus07], [CJ10], [NF12], [FR14c], [FNR14], [BPT15] and [Bra16] (see Table 6.1). Moreover, we analyzed whether popular proxy caches as well as CDNs do store and reuse error codes returned from the origin server. This exploratory study has been conducted with the approach of Nguyen et al. [NLF19a; NLF18]. They provide a freely available cache testing tool for analyzing web browser caches, proxy caches and CDNs in a systematically manner. The cache testing tool also offers a test suite containing 397 test cases that can be customized by a test case specification language. We extended the suite by adding new tests for evaluating the caching of responses containing error status codes. In our study we concentrated on the five well-known proxies caches Apache HTTP Server (Apache HTTPD) v2.4.18, Nginx v1.10.3, Varnish v6.0.1, Apache Traffic Server (Apache TS) v8.0.2 and Squid v3.5.12 as well as the CDNs Akamai, CloudFront, Cloudflare, Stackpath, Azure, CDN77, CDNSun, Fastly, KeyCDN and G-Core Labs.

Even though the cacheability of error codes are well-defined by the series of RFC specifications given above, our analysis reveals that some web caching systems violates some of these policies. For instance, CloudFront and Cloudflare do store and reuse error messages such as `400 Bad Request`, `403 Forbidden` and `500 Internal Server Error` although being not permitted. The violation of web caching policies is a severe issue and needs to be taken into account by content providers and web caching system vendors. Recent publications have revealed that non-adherence may otherwise lead to caching vulnerabilities [Gil17; Lin+05; Che+16]. Following these observations, we investigated further in order to discover vulnerable constellations. We were able to identify three concrete instantiations of the general CPDoS attack that we present in the following subsections.

#### 6.4.1 HTTP Method Override (HMO) Attack

The HTTP standard [FR14c] defines a set of request methods for the client to indicate the desired action to be performed for a given resource. GET, POST, DELETE, PUT and PATCH



Legend: ✓ cacheable status code according to HTTP Standard, ● stored by web caching system, ○ not stored by web caching system, ■ storing not cacheable status code

Error Code	Cacheable	Apache HTTPD	Apache TS	Nginx	Squid	Varnish	Akamai	Azure	CDN77	CDN5un	Cloudflare	CloudFront	Fastly	G-Core Labs	KeyCDN	Stackpath
400 Bad Request	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
401 Unauthorized	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
402 Payment Required	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
403 Forbidden	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
404 Not Found	✓	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○
405 Method Not Allowed	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
406 Not Acceptable	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
407 Proxy Authentication Required	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
408 Request Timeout	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
409 Conflict	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
410 Gone	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
411 Length Required	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
412 Precondition Failed	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
413 Payload Too Large	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
414 Request-URI Too Long	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
415 Unsupported Media Type	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
416 Requested Range Not Satisfiable	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
417 Expectation Failed	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
418 I'm a teapot	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
421 Misdirected Request	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
422 Unprocessable Entity	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
423 Locked	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
424 Failed Dependency	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
426 Upgrade Required	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
428 Precondition Required	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
429 Too Many Requests	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
431 Request Header Fields Too Large	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
444 Connection Closed Without Response	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
451 Unavailable For Legal Reasons	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
499 Client Closed Request	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
500 Internal Server Error	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
501 Not Implemented	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
502 Bad Gateway	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
503 Service Unavailable	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
504 Gateway Timeout	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
505 HTTP Version Not Supported	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
506 Variant Also Negotiates	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
507 Insufficient Storage	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
508 Loop Detected	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
510 Not Extended	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
511 Network Authentication Required / Status Code and Captive Portals	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
599 Network Connect Timeout Error	-	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

Table 6.1: Overview of cacheable error status codes according to [HM98; Mas98; NL00; Dus07; CJ10; NF12; FR14c; FNR14; BPT15; Bra16] and empirical study results showing whether the status codes are cached by the analyzed web caching systems

are arguably the most used HTTP methods in web applications and REST-based web services [RR08] in particular. Some intermediate systems such as proxies, load balancer, caches or firewalls, however, only support GET and POST. This means DELETE, PUT and PATCH requests are simply blocked. To circumvent this restriction many REST-based APIs or web frameworks provide auxiliary headers such as X-HTTP-Method-Override, X-HTTP-Method or X-Method-Override for passing through an unrecognized HTTP method. These headers will usually be forwarded by any intermediate systems. Once the request reaches the server, a method override header instructs the web application to replace the method in the request line with the one in the method overriding header value.

These method override headers are very useful in scenarios when intermediate systems block distinct HTTP methods. However, if a web application supports such a header and also uses a shared web caching system, a malicious client can exploit this semantic gap for performing a CPDoS attack. In a typical HTTP Method Override (HMO) attack flow, a malicious client crafts a GET request including an HTTP method overriding header as shown in Figure 6.3.

A CDN or reverse proxy cache interprets the request in Figure 6.3 as a benign GET request targeting `http://example.org/index.html`. Hence, it forwards the request with the X-HTTP-Method



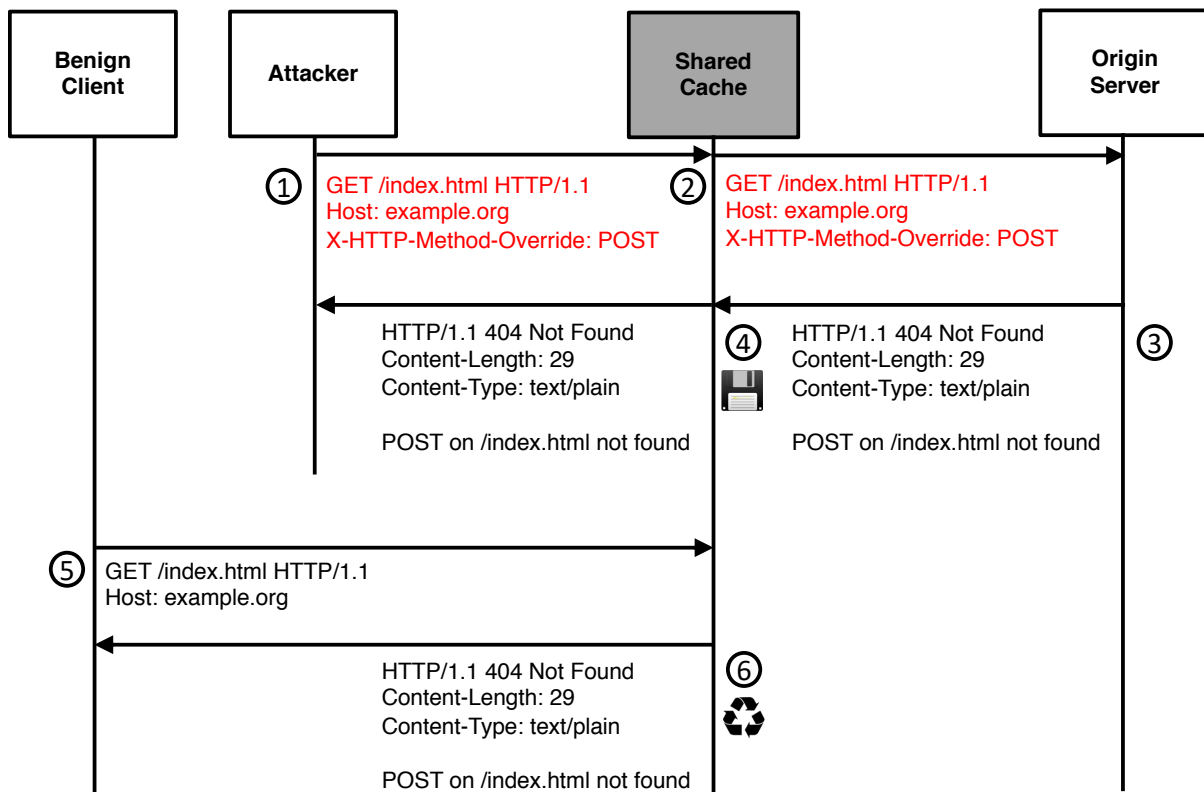


Figure 6.3: Flow and example construction of the HTTP Method Override (HMO) attack

-Override header to the origin server. The endpoint, however, interprets this request as a POST request, since the `X-HTTP-Method-Override` header instructs the server to replace the HTTP method in the request line with the one contained in the header. Accordingly, the web application returns a response based on POST. Let's assume that the target web application does not implement any POST endpoint for `/index.html`. In such a case, web frameworks usually returns an error message, e.g., the status code `404 Not Found` or `405 Method Not Allowed`. The shared cache assigns the returned response with the error code to the GET request targeting `http://example.org/index.html`. Since the status codes `404 Not Found` and `405 Method Not Allowed` are cacheable according to the HTTP Caching RFC 7231 as shown in Table 6.1, caches store and reuse this error response for recurring requests. Each benign client making a subsequent GET request to `http://example.org/index.html` receives the cached error message instead of the legitimated web application's start page.

#### 6.4.2 HTTP Header Oversize (HHO) Attack

The HTTP standard does not define any size limit for request headers. Hence, intermediate systems, web servers and web frameworks specify their own limit. Most web servers and proxy caches provide a request header limit of about 8,000 bytes in order to avoid security threats such as *request header overflow* [NAT10] or *ReDoS* [SP18] attacks. However, there are also intermediate systems, which specify a limit larger than 8,000 bytes. For instance, the Amazon CloudFront CDN allows up to 24,713 bytes. In an exploratory study we gathered the default HTTP request header limits deployed by various HTTP engines and cache systems (see Table 6.3).

This semantic gap in terms of different request header size limits can be exploited to conduct a CPDoS attack. To execute an HTTP Header Oversize (HHO) attack, a malicious client needs to send a GET request including a header larger than the limit of the origin server but smaller than the one of the cache. To do so, an attacker has two options. First, she crafts a request header with many malicious headers. The other option is to include one single header with an oversized key or value as shown in Figure 6.4.

The web caching system forwards this request including the oversized header to the endpoint, since the header size is under the limit of the intermediary. The web server, however, blocks this request and returns an error page, as the request exceeds the header size limit. This returned error page is stored and will be reused for equivalent requests.

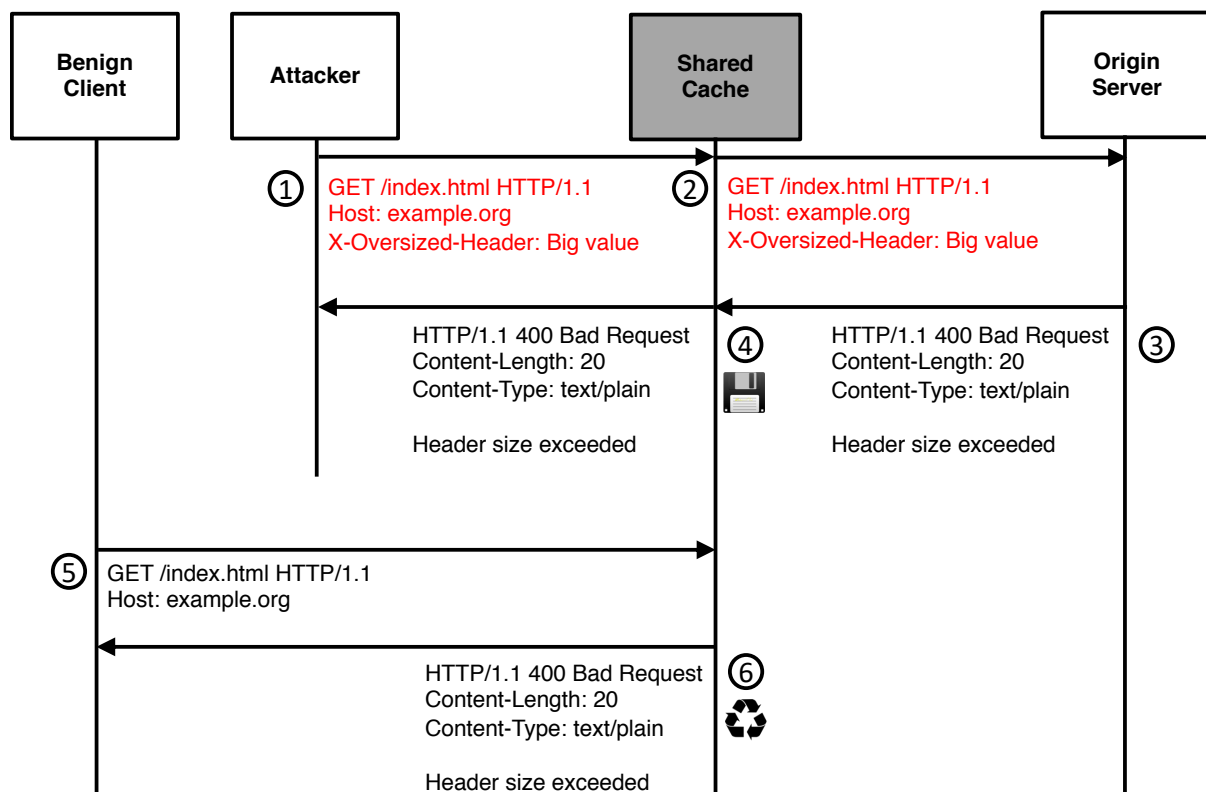


Figure 6.4: Flow and example construction of the HTTP header oversize (HHO) attack

### 6.4.3 HTTP Meta Character (HMC) Attack

The HTTP Meta Character (HMC) works similar to the HHO attack. Instead of sending an oversized header, this attack tries to bypass a cache with a request header containing a harmful meta character. Meta characters can be e.g. control characters such as the line break/carriage return (`\n`), line feed (`\r`) or any other Unicode control characters. As the `\n` and `\r` characters are used by the response splitting attack to poison a cache, some HTTP implementations block requests containing these symbols.

HTTP implementations, which drop such characters, mostly return an error message signaling that they do not parse this request. However, there are some cache intermediaries which do not care about certain control characters. They simply forward the request including the meta character to the origin server which return an error code. The resulting error page is then stored

and reused by the cache. This constellation can be exploited by a malicious client to conduct another form of CPDoS attack. We declare this vulnerability as HTTP Meta Character (HMC) attack. To do so, the attacker crafts a request with a meta character, e.g. `\n`, as shown in Figure 6.5. The goal of this example attack is to fool the origin server into believing that it is attacked by a response splitting request. As with the previously presented vulnerabilities, the HMC request traverses the cache without any issues. Once the request reaches the endpoint, it is blocked and an according error page is returned, since the web server is aware of the implications regarding suspicious characters such as `\n`. This error message is then stored and recycled by the corresponding web caching system.

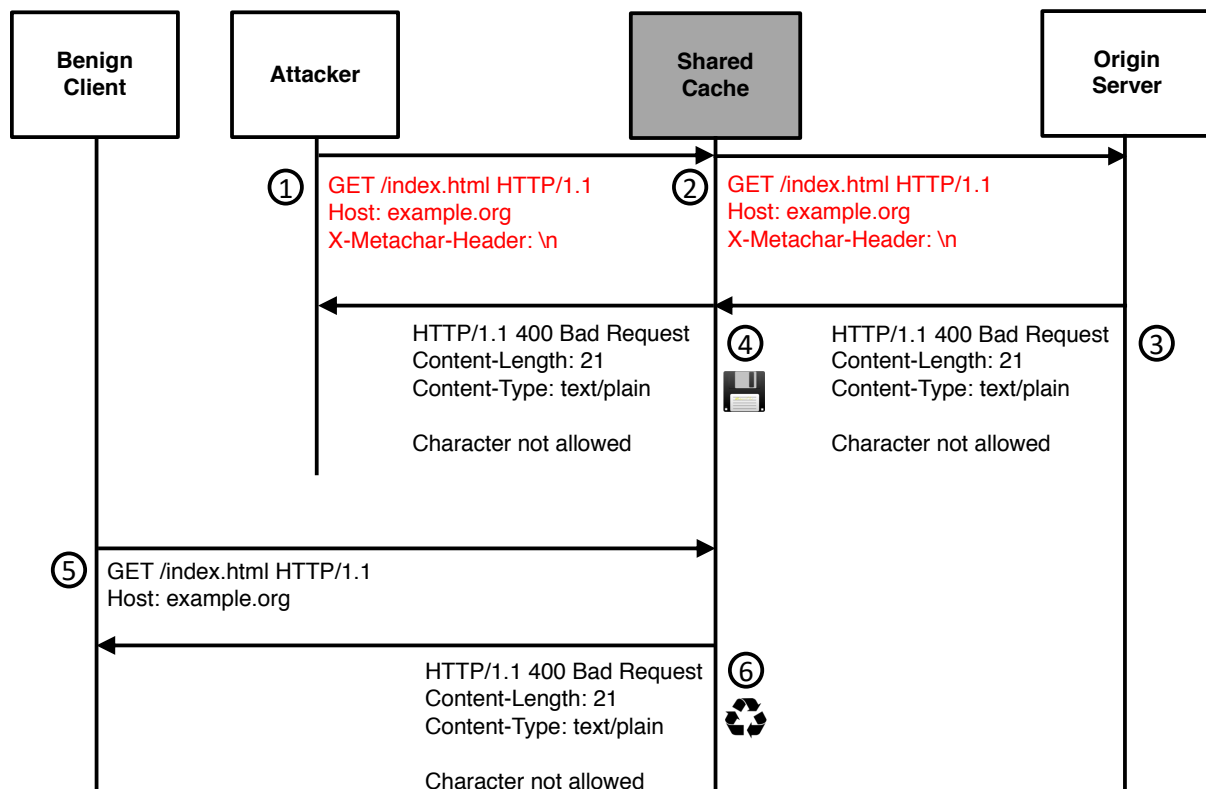


Figure 6.5: Flow and example construction of the HTTP Meta Character (HMC) attack

## 6.5 Practicability of CPDoS Attacks

In order to explore the existence of CPDoS weaknesses in the wild, we conducted a series of experiments. A crucial prerequisite for a potential CPDoS vulnerability is a web caching system that stores and reuses error pages produced by the origin server. Table 6.1 highlights that Varnish, Apache TS, Akamai, Azure, CDN77, Cloudflare, CloudFront and Fastly do so. Based on these findings, we conducted three experiments—one for each introduced CPDoS variant—to examine whether these intermediate systems are vulnerable to CPDoS attacks.

### 6.5.1 Experiments Setup

The first step to analyze whether CPDoS vulnerabilities exist in practical environments is to figure out vulnerable HTTP implementations which are utilized as the origin server. HTTP

implementations on the origin server can be diverse systems including, e.g., reverse proxies, web servers, web frameworks, cloud services or other intermediate systems as well as another cache.

In our first experiment, we analyzed the method override header support in web frameworks. Additionally, we also evaluated what error page is returned when sending a method override header containing an HTTP method which is not implemented by corresponding resource endpoint. Based on the findings in Table 6.1 where we know what error page is stored by what web caching systems, we inferred what web framework in combination with what web caching systems might be vulnerable to HMO attacks. For this empirical analysis we chose 13 web frameworks based on the most popular programming languages according to IEEE Spectrum [IEE18]. The analyzed collection of web frameworks includes ASP.NET v2.2, BeeGo v1.10.0, Django v2.1.7, Express.js v4.16.4, Flask v1.0.2, Gin v1.3.0, Laravel v5.7, Meteor.js v1.8, Rails v5.2.2, Play Framework 1 (Play 1) v1.5.1, Play Framework 2 (Play 2) v2.7, Spring Boot v2.1.2 and Symfony v4.2.

The second experiment investigated the request header size limits of the web caching systems in Table 6.1 as well as the 13 web frameworks. As the web frameworks ASP.NET and Spring Boot requires an underlying web server to be deployed in production mode, we additionally also evaluate the request header limits of Microsoft Internet Information Services (IIS) v10.0.17763.1 and Tomcat v9.0.14. Moreover, we also evaluated popular cloud services including Amazon S3, Github Pages, Gitlab Pages, Google Storage and Heroku. As with the first experiment, we also tested which error code is returned when the request header size limit is exceeded. With these findings we figured out what HTTP implementations in conjunction with what web caching systems are potentially vulnerable to HHO attacks.

The last experiment evaluated the feasibility of HMC attacks. Here, we evaluated the handling of meta characters in all mentioned web caching systems, web frameworks, web servers and cloud services. To test as many meta characters as possible we collected a list of 520 potentially irritating strings. This collection contains control, special, international and other unicode characters as well as strings comprising attack vectors including cross site scripting (XSS), SQL injections and remote execution attacks. The goals of this study was to analyze what characters and strings are blocked, sanitized and processed or forwarded without any issues. Moreover, we also evaluated what error page is triggered when a character or string is blocked. Based on our findings we were able to conclude what characters and what symbols need to be send to what constellation of HTTP engine and web caching system to induce an HMC attack.

### 6.5.2 Feasibility of HMO attacks

Table 6.2 shows the results of the first experiment. It highlights that Symfony, Laravel and Play 1 support method override headers by default. Django and Express.js instead do not consider method override headers by default, but provide plugins to add this feature. Flask does not offer any plugin for the integration of method override headers, but provides an official tutorial how to enable it [Fla10]. Table 6.2 also points out what error code is returned when the web framework receives a method override header with an action that is not implemented by the addressed resource endpoint.

Even though the web frameworks with a method overriding header support return cacheable error codes, we observed that only Play 1 and Flask are vulnerable to HMO CPDoS attacks.

Legend: ○ must be implemented manually, ● by default, ◐ not by default but by extension

Web framework	Programming lang.	Method overriding support	Error code when method not implemented
Rails	Ruby	○	undefined
Django	Python	◐	405
Flask	Python	◐	405
Express.js	JavaScript	◐	405
Meteor.js	JavaScript	○	undefined
BeeGo	Go	○	undefined
Gin	Go	○	undefined
Play 1	Java	●	404
Play 2	Java/Scala	○	undefined
Spring Boot	Java	○	undefined
Symfony	PHP	●	405
Laravel	PHP	●	405
ASP.NET	C#	○	undefined

Table 6.2: HTTP method overriding headers support of tested web frameworks

However, both web frameworks can only be affected if Fastly, Akamai, Cloudflare, CloudFront, CDN77 and Varnish are used as intermediate cache. The reason why these web frameworks are vulnerable lies in the fact that Play 1 and Flask do perform an HTTP method change for GET as well as POST requests in case an HTTP method override header is present. Laravel, Symfony and the plugins for Django and Express.js are not vulnerable to HMO CPDoS, since they ignore HTTP method override headers in GET requests and restrict themselves to transform the method for POST requests only. Attackers cannot poison the tested web caching systems with a POST request, since responses to POST requests are not stored by any of them.

Malicious clients can attack web applications implemented with the Play 1 by sending a GET request with the method override header including, e.g., POST as value. If the corresponding resource endpoint does not implement any functionality for POST, then the web framework returns the error code 404 Not Found. Akamai, Fastly, CDN77, Cloudflare, CloudFront and Varnish cache this status code by default (see Table 6.1). Flask is also vulnerable to HMO CPDoS attacks, if the support of HTTP method override headers is implemented with the official tutorial of the web framework's website. However, HMO attacks are only possible, if Akamai and CloudFront are utilized as CDN, since Flask returns the status code 405 Method Not Allowed. Akamai and CloudFront are the only analyzed web caching systems, which store and reuse error pages with this code.

### 6.5.3 Feasibility of HHO attacks

Table 6.3 depicts the results of our study on request header size limits. If available, it moreover lists the request header size limit specified in the documentation of the corresponding HTTP implementation. Note, that we omit the web frameworks ASP.NET, Django, Flask, Laravel, Rails, Symfony and Spring Boot in this table, as we found out that the request header limits depend on the used web server and deployment environment.

Our obtained results reveal many varieties in terms of request header size limits among the HTTP implementations. The evaluation shows that CloudFront provides a request header size limit, which is much higher than the one of the many other HTTP implementations we tested. Moreover, Amazon's CDN also caches the error code 400 Bad Request by default (see Table 6.1), which is triggered by most of the HTTP implementations when the request header size limit is exceeded. Hence, in our experiments we figured out that when using CloudFront as CDN any HTTP implementation that has a request header size limit lower than CloudFront

	HTTP implementation	Documented limit	Tested limit	Limit exceed error code	
CDN	Akamai	undefined	32,760 bytes	No Response	
	Azure	undefined	24,567 bytes	400	
	CDN77	undefined	16,383 bytes	400	
	CDNSun	undefined	16,516 bytes	400	
	Cloudflare	undefined	≈ 32,395 bytes	400	
	Cloudfront	20,480 bytes	≈ 24,713 bytes	494	
	Fastly	undefined	69,623 bytes	No Response	
	G-Score Labs	undefined	65,534 bytes	400	
	KeyCDN	undefined	8,190 bytes	400	
	StackPath	undefined	≈ 85,200 bytes	400	
	HTTP engine	Apache HTTPD	8,190 bytes	8,190 bytes	400
		Apache HTTPD + ModSecurity	undefined	8,190 bytes	400
		Apache TS	131,072 bytes	65,661 bytes	400
Nginx		undefined	20,584 bytes	400	
Nginx + ModSecurity		undefined	8,190 bytes	400	
IIS		undefined	16,375 bytes	400, (404)	
Squid		65,536 bytes	65,527 bytes	400	
Tomcat		undefined	8,184 bytes	400	
Varnish		8,192 bytes	8,299 bytes	400	
Cloud Service		Amazon S3	undefined	≈ 7,948 bytes	400
		Github Pages	undefined	8,190 bytes	400
	Gitlab Pages	undefined	>500,000 bytes	undefined	
	Google Cloud Storage	undefined	16,376 bytes	413	
	Heroku	8,192 bytes	8,154 bytes	400	
	Web Framework	BeeGo	undefined	>500,000 bytes	undefined
Express.js		undefined	81,867 bytes	No Response	
GIN		undefined	>500,000 bytes	undefined	
Meteor.js		undefined	81,770 bytes	400	
Play 1		undefined	8,188 bytes	No Response	
Play 2		8,192 bytes	8,319 bytes	400	

Table 6.3: Request header size limits of HTTP implementations

and returns the status code 400 Bad Request if the limit is exceeded is vulnerable to HHO CPDoS attacks. For instance, the web caching systems Apache HTTPD and Nginx, which can also be used as web server or reverse proxy provide a lower request header size limit than CloudFront.

Besides the fact that Apache HTTPD and Nginx are amongst the most used web servers according to a survey of Netcraft [Net19], both systems are often deployed with other intermediate systems. When using one of these HTTP implementations in conjunction with CloudFront, these systems can be affected by an HHO CPDoS attack. This also means if Apache HTTPD and Nginx is configured as intermediate reverse proxy in front of other web applications, then these systems are vulnerable to HHO CPDoS as well. Moreover, Apache HTTPD and Nginx are often utilized as web server and deployment environment for web frameworks such as Rails, Django, Flask, Symfony and Laravel. All these web frameworks are vulnerable to HHO CPDoS likewise if they are deployed with Apache HTTPD or Nginx. Spring Boot and ASP.NET can also be affected by HHO CPDoS attacks, as both web frameworks require a web server in production mode. Spring Boot can be deployed with Tomcat and ASP.NET can use IIS as the underlying deployment environment. Tomcat and IIS have request header size limits lower than CloudFront. Both web servers return the error 400 Bad Request for oversized header likewise. The cloud service Heroku is another deployment platform for web frameworks. It supports, e.g., Django, Flask, Laravel, Rails, Laravel and Symfony. As Heroku provides a request header size limit lower than CloudFront, web applications using the cloud service in conjunction with the CDN can be vulnerable as well. Other HTTP implementations which can be affected by HHO CPDoS attacks when using CloudFront as CDN are Play 2 as well as the cloud services Amazon S3, Github Pages and Heroku. Play 1 is also vulnerable to HHO CPDoS attacks, even though it does not return an error page when the request header size limit is exceeded. The web framework does not return any response if it receives an oversized header. Here, the TCP socket remains open until the web application shuts down. If CloudFront notices such an idle communication channel, then the CDN returns the error code 502 Bad Gateway. This error message is stored and reused for recurring requests likewise. According to our experiments, Google storage in conjunction with CloudFront is not vulnerable to HHO CPDoS although the cloud service



has lower request header size limit than the CDN. Google storage returns the error code `413 Payload Too Large` for oversized headers and this error message is not cached by any of the analyzed web caching systems. Table 6.3 also contains a result obtained when using Nginx with the WAF plugin ModSecurity. In such a configuration, conducting a successful HHO CPDoS attack is even easier as without the security extension. The tested request header limit of Nginx is around 20,000 bytes but when ModSecurity is added to both systems, it reduces the restriction to 8,190 bytes. Even though the usage of ModSecurity should actually avoid web application attacks such as DoS, it eases to conduct an HHO CPDoS attack in this case.

As mentioned before, IIS and web frameworks such as APS.NET running on this web server are vulnerable to HHO CPDoS attacks when using CloudFront as CDN. However, in certain circumstances, they might also be vulnerable when Akamai, Fastly, CDN77, Cloudflare and Varnish are utilized. The IIS web server provides an option to set a size limit for a distinct request header. Some web applications require such a configuration option to block, e.g., an oversized `Cookie` header. If this restriction is defined for a request header and this limit is exceeded, then the web server return the error code `404 Not Found`. This error message is cached by Akamai, Fastly, CDN77, CloudFront, Cloudflare and Varnish.

#### 6.5.4 Feasibility of HMC attacks

Table 6.4 shows the results of our third experiments where we analyzed the handling of strings containing meta characters. For the sake of readability, we only list the characters and strings that are blocked or sanitized by at least one of the tested HTTP implementations. Moreover, we omit the web frameworks ASP.NET, Django, Flask, Laravel, Spring Boot and Symfony in this table, since the handling of meta characters depends on the used web server and deployment environment.

The evaluation highlights that the many analyzed systems consider control characters as a threat. Suspicious characters or strings are either blocked by the denoted error code or are sanitized from the request header. However, the handling of meta strings and characters are very diverse. For instance, CloudFront blocks the character `\u0000` and sanitizes `\n`, `\v`, `\f`, `\r`, but forwards other control characters such as `\a`, `\b` and `\e` without modifying them. If Apache HTTPD, IIS or Varnish is used with CloudFront, then the corresponding systems block the forwarded header containing forbidden characters with the status code `400 Bad Request`. CloudFront stores such an error message. This means when using CloudFront as CDN, all tested HTTP implementations, which blocked harmful strings and characters that are not rejected or sanitize by CloudFront, are vulnerable to HMC CPDoS attacks. Besides Apache HTTPD, IIS and Varnish, this includes Github Pages, Gitlab Pages, BeeGo, Gin, Meteor.js and Play 2. Express.js is vulnerable to HMC CPDoS attacks as well, even though it does not block any tested string by an error code. The issue here is similar to the problem of oversized header in Play 1. When sending a request header with multiple control characters Express.js does not reply at all. Accordingly, CloudFront returns the error message `502 Bad Gateway` to the client. This error code is also stored and reused for subsequent requests.

#### 6.5.5 Consolidated Review of Analysis Results

Based on our findings of all three experiments, we detected many CPDoS attack vectors in various different combinations of web caching systems and HTTP implementations. Most of



Legend: ○ processed/forwarded without error and sanitization

Meta character in request header	Akamai	Azure	CDN77	CDNSun	Cloudflare	Cloudfront	Fastly	G-Score Labs	KeyCDN	Stackpath
\u0000	400	400	400	400	400	400	No Response	400	400	Sanitized
\u0001... \u0006	○	400	Sanitized	○	○	○	400	○	○	○
\a	○	400	Sanitized	○	○	○	400	○	○	○
\b	○	400	Sanitized	○	○	○	400	○	○	○
\t	○	○	○	○	○	○	○	○	○	○
\n	○	400	Sanitized	Sanitized	Sanitized	Sanitized	Sanitized	Sanitized	○	Sanitized
\v	○	400	Sanitized	○	○	Sanitized	400	○	○	Sanitized
\f	○	400	Sanitized	○	○	Sanitized	400	○	○	Sanitized
\r	○	400	Sanitized	○	Sanitized	Sanitized	400	Sanitized	○	Sanitized
\u000e... \u001f, \u007f	○	400	Sanitized	○	○	○	400	○	○	○
Multiple Unicode control character (e.g. \u0001\u0002)	○	400	Sanitized	○	○	○	400	○	○	○
() {0;}; touch /tmp/blns.shellshock1.fail;	○	○	○	○	403	○	○	○	○	○
() { _; } >_[\${\$()}] { touch /tmp/blns.shellshock2.fail; }	○	○	○	○	403	○	○	○	○	○
Meta character in request header	Apache HTTPD + (ModSecurity)	Apache TS	Ngix + (ModSecurity)	IIS	Tomcat	Squid	Varnish	Amazon S3	Google Storage	
\u0000	400	400	400	400	○	○	400	○	○	
\u0001... \u0006	400	○	○	400	○	○	400	○	○	
\a	400	○	○	400	○	○	400	○	○	
\b	400	○	○	400	○	○	400	○	○	
\t	○	○	○	400	○	○	400	○	○	
\n	400	○	Sanitized	○	○	○	Sanitized	○	○	
\v	400	○	○	400	○	○	400	○	○	
\f	400	○	○	400	○	○	400	○	○	
\r	400	○	○	400	○	○	400	○	○	
\u000e... \u001f, \u007f	400	○	○	400	○	○	400	○	○	
Multiple Unicode control character (e.g. \u0001\u0002)	400	○	○	400	○	○	400	○	○	
() {0;}; touch /tmp/blns.shellshock1.fail;	○	○	○	○	○	○	○	○	○	
() { _; } >_[\${\$()}] { touch /tmp/blns.shellshock2.fail; }	○	○	○	○	○	○	○	○	○	
Meta character in request header	Github Pages	Gitlab Pages	Heroku	Beego	Express.js	Gin	Meteor	Play 1	Play 2	
\u0000	No Response	400	○	400	○	400	400	○	400	
\u0001... \u0006	400	400	○	400	○	400	400	○	400	
\a	400	400	○	400	○	400	400	○	400	
\b	400	400	○	400	○	400	400	○	400	
\t	400	○	○	○	○	○	○	○	○	
\n	400	○	400	○	○	○	○	○	○	
\v	400	400	○	400	○	400	400	○	400	
\f	400	400	○	400	○	400	400	○	400	
\r	400	400	○	400	○	400	○	○	400	
\u000e... \u001f	400	400	○	400	○	400	400	○	400	
\u007f	400	400	○	400	○	400	400	○	○	
Multiple Unicode control character (e.g. \u0001\u0002)	400	400	○	400	No Response	400	No Response	○	400	
() {0;}; touch /tmp/blns.shellshock1.fail;	○	○	○	○	○	○	○	○	○	
() { _; } >_[\${\$()}] { touch /tmp/blns.shellshock2.fail; }	○	○	○	○	○	○	○	○	○	

Table 6.4: Meta string handling in request header of HTTP implementations

the attacks are executable on CloudFront as shown in Table 6.5. This overview summarizes what pair of web caching system and HTTP implementation is vulnerable to what CPDoS attack. The experiments' results show that web applications using CloudFront are highly vulnerable to CPDoS attacks, since the CDN caches the error code 400 Bad Request by default. Many server-side HTTP implementations return this error message when sending a request with an oversized header or meta characters. The likelihood to be affected by CPDoS attacks when utilizing the other analyzed caches including Varnish, Akamai, CDN77, Cloudflare or Fastly is rather lower. These web caching systems do store the error code 404 Not Found but not 400 Bad Request. The caching of error pages with status code 404 Not Found is a proper and compliant approach for optimizing website performance. In this case, there is no malfunction in Varnish, Akamai, CDN77, Cloudflare and Fastly. The reason for a successful CPDoS attack lies in the fact that, Play 1 and Microsoft IIS allows to provoke 404 Not Found error pages on resource endpoints which do not return an error message when sending a benign request.

Legend: ○ no CPDoS attack detected

Apache HTTPD	Apache TS	Nginx	Squid	Varnish	Akamai	Azure	CDN77	CDNSun	Cloudflare	CloudFront	Fastly	G-Score Labs	KeyCDN	StackPath	Web caching system / Origin server HTTP implementation
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Apache HTTPD + (ModSecurity)
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Apache TS
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Nginx + (ModSecurity)
○	○	○	○	(HHO)	(HHO)	○	(HHO)	○	(HHO)	HHO, HMC	(HHO)	○	○	○	IIS
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Tomcat
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Squid
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Varnish
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Amazon S3
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Google Cloud Storage
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Github Pages
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Gitlab Pages
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Heroku
○	○	○	○	(HHO)	(HHO)	○	(HHO)	○	(HHO)	(HHO), (HMC)	(HHO)	○	○	○	ASP.NET
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	BeeGo
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Django
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Express.js
○	○	○	○	(HMO)	(HMO)	○	○	○	○	HMO, (HHO), (HMC)	○	○	○	○	Flask
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Gin
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Laravel
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Meteor.js
○	○	○	○	HMO	HMO	○	HMO	○	HMO	HHO, HMO	HMO	○	○	○	Play 1
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Play 2
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Rails
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Spring Boot
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Symfony

Table 6.5: CPDoS vulnerability overview

### 6.5.6 Practical Impact

In the first step to estimate the practical impact of CPDoS attacks, we determined the amount of websites that use one of the vulnerable web caching systems and HTTP implementations listed in Table 6.5. Our approach to find vulnerable real world websites is to inspect the response header.

Many HTTP implementations append informational headers to the response for declaring that a message is processed by this entity. For instance, CloudFront includes the values `Hit` from CloudFront or `Miss` from CloudFront to the `x-cache` header and Microsoft IIS adds the string `Microsoft-IIS` to the `Server` header. By means of this information an attacker can unambiguously detect what cache or what server-side HTTP implementation is used by the target web application respectively. Based on this approach, we analyzed the websites of the U.S. Department of Defense (DoD)<sup>1</sup> and the Alexa Top 500 websites. In addition to this, we used the Google Big Query service to investigate over 365 million URLs stored in the HTTP Archive data set `httparchive.summary_requests.2018_12_15_desktop`. Table 6.6 shows the number of websites and URLs of the DoD, the Alexa Top 500 and the HTTP Archive where the response header indicates that the content is processed by a vulnerable HTTP implementation.

The results highlight that eight websites of the DoD, 23 of the Alexa Top 500 and over twelve million URLs stored in the mentioned data set of the HTTP Archive are served via CloudFront. Moreover, all eight websites of the DoD, 16 websites of the Alexa Top 500 and over nine million URLs of the HTTP Archive point out that CloudFront in combination with Apache HTTPD, Nginx, Amazon S3, Microsoft IIS and Varnish is used. Our experiments revealed that these constellations are vulnerable to CPDoS attacks (see Table 6.5). However, it is very difficult to estimate the exact number of vulnerable websites without inspecting each of them individually.

1. <https://dod.defense.gov/About/Military-Departments/DoD-Websites/>

	DoD	Alexa Top 500	HTTP Archive
Total number of web sites/URLs	414	500	365.112.768
Varnish	2	40	4.658.950
Akamai	2	38	1.031.535
CDN77	0	0	321.456
Cloudflare	7	34	18.236.800
CloudFront	8	23	12.140.461
Fastly	0	9	4.013.578
IIS	27	9	17.792.692
Flask	0	0	5.765
Play 1	0	0	10.491

Table 6.6: Number of websites/URLs using Varnish, Akamai, CDN77, Cloudflare, CloudFront, Fastly, IIS, Flask and Play 1

Moreover, the experiments have been done with the default configuration and without taking any other intermediate system into account. It is, however, very common that content providers change the default configuration of a cache in order to adapt the caching policy to the respective needs. Moreover, real world web applications also utilize other intermediate systems such as load balancers or WAFs. All these settings influence the practicability of CPDoS attacks in any direction. To get a clearer picture on the real life impact of CPDoS attacks, we took some samples based on the URLs from the Alexa Top 500, DoD, and HTTP Archive data sets. Overall, we found twelve vulnerable resources within a few days. These also include mission-critical websites such as `ethereum.org`, `marines.com`, and `nasa.gov` which use CloudFront as CDN. At all these websites, we were able to block multiple resources including scripts, style sheets, images, and even dynamic content such as the start page. The visual damage of a CPDoS attack is shown by the Figures 6.6 and 6.7 in the Appendix A. In Figure 6.6, the CPDoS attack is first applied to an image referenced in the start page of the victim website `ethereum.org`. Then the style sheet file is denied and finally, an error page replaces the whole start page. Figure 6.7 illustrates the affected start page of `marines.com` which displays an error page to the user instead of the genuine content. Moreover, we were also able to conduct a successful CPDoS attack on the update files of IKEA’s Smart Home devices. IKEA uses CloudFront in conjunction with S3 to distribute remote control firmware and driver updates for their wireless bulbs. As CloudFront in combination with S3 is vulnerable to HHO CPDoS attacks, an attacker can block the remote control devices of IKEA from fetching security patches. These evidences show that CPDoS attacks can affect static as well as dynamic resources. Most of the vulnerable websites use CloudFront as CDN. However, the real world impact of CPDoS attacks is not only bound to CloudFront. We also found vulnerable websites in our sample which utilize other CDNs such as Akamai or Cloudflare in conjunction with Play 1. We have uncovered these examples in a few days only. An advanced attacker with political and financial motivation is easily able to gather much more vulnerable resources as they only need to investigate the response headers in order to estimate whether a target website or resource is potentially vulnerable to CPDoS attacks. Moreover, the freely available HTTP Archive data sets via Google Big Query include millions of URLs which can be investigated by an attacker. For instance, HTTP Archive data set `httparchive.summary_requests.2018_12_15_desktop` contains over 9 millions URLs which we considered as highly vulnerable since the response headers of these resources indicate that CloudFront in conjunction with Apache HTTPD, Nginx, Amazon S3, Microsoft IIS, and Varnish is used. Among them are also many critical websites and resources including

Amazon itself, the website `dowjones.com`, as well as Logitech which distributes firmware via CloudFront.

### 6.5.7 Practical Considerations

Caches are only vulnerable to CPDoS attacks if they store and reuse error pages. Web caching systems such as Stackpath, CDNSun, KeyCDN and G-Core labs cannot be affected by CPDoS attacks, since these CDNs do not cache error messages at all. This is also true for Apache HTTPD, Nginx and Squid when using them as an intermediate cache without involving any other vulnerable web caching systems.

As with other cache poisoning vulnerabilities, CPDoS attacks are only possible when a vulnerable web caching system does not contain a fresh copy of the to be attacked resource. That is, if a shared cache still maintains and reuses a stored fresh response for recurring requests, a malicious request is not able to poison the intermediary. The web caching system serves all requests to the target resource. None of the requests are forwarded to the origin server until the freshness lifetime is expired, so that no error page can be triggered. This means if a cache still owns a fresh response, an attacker has to wait until the cached content is stale. The most straightforward information to find out the expiration time is the `Expires` header which indicates the absolute expiration date. If the response does not contain an `Expires` header or the expiration time of this header is overridden by the `max-age` or `s-maxage` directive control directive, the attacker can make use of the `Age` header. The `Age` header declares the seconds of stay in the cache. The value of the `Age` header subtracted from the value of the `max-age` or `s-maxage` directive is the relative expiration time of the cached response. If the cached response is expired, the attacker's request must be the very first request so that it can reach the origin server to trigger an error page. To increase the likelihood for being the first request, we send automatized requests with a one second interval when the response is close to expire. With this technique we were able to successfully attack all twelve vulnerable websites of our spot check experiment. Sending regularly performed requests with one second distance of time is also a useful approach for cached responses which does contain any expiration time information, i.e., resources which are implicitly cached. Such responses usually do not contain any `max-age` or `s-maxage` directives and `Expire` headers. Here, the attacker needs to send automatized requests until one of the requests is forwarded to the origin server. Moreover, automatized requests with a one second interval are not considered as harmful even when they are sent over a long time, since health checks requests can also have the same interval. We tested this technique on several CDNs which also included WAFs and DDoS protections. Since we only used a single client to perform the attack, none of CDNs detected the malicious requests.

Many web applications configure the proxy cache or the CDN to serve the whole website. This means all resources including dynamic pages and static files are forwarded and processed by the cache. To exclude dynamic pages from being implicitly cached, content providers include `no-store` or `max-age=0` to the response header, so that each request must be forwarded to the origin server. If a vulnerable cache in conjunction with a vulnerable server-side HTTP implementation is used, these resources can be attacked without the need to wait and any automation of sending requests. One single malicious request is enough to paralyze the target resource, since each request is forwarded to the origin server. Vulnerable websites which configure the CDN to serve all resources are, e.g., `marines.com`, `ethereum.org` and `nasa.gov`.

There also many web applications which only configure the cache to store and reuses responses of certain URL paths such as for static files in the javascript or images directory. Other URL paths are accordingly not cached at all. Many content providers also maintain subdomains (e.g. static.example.org) or a specific domain for static files which are served via a cache. In these cases, only resources within the cached URL paths or the specific domain can be affected. To find out whether a distinct response traverses a cache, an attacker can inspect the response headers. For instance, the `Age` response header indicate that a cache is utilized. The main website of IKEA (ikea.com) does not use CloudFront or any other vulnerable HTTP implementations which indicates that this homepage is most likely not vulnerable to CPDoS attacks. However, IKEA uses a specific domain (fw.ota.homesmart.ikea.net) in conjunction with CloudFront to host the update files of their Internet of Things devices.

Another important limitation of CPDoS attacks is that the web caching systems except Fastly do only cache error pages for few minutes or seconds. Fastly stores and reuses the error page for one hour. If this time span is over, then the first benign request to the target resource is forwarded to origin server and refreshed again. Still, to extend the duration of CPDoS attacks, malicious clients can resend harmful requests in accordance to the fixed interval.

## 6.6 Responsible disclosure

All discovered vulnerabilities have been reported to the HTTP implementation vendors and cache providers on February 19, 2019. We worked closely with these organizations to support them in eliminating the detected threats. We did not notify the website owners directly, but left it to the contacted entities to inform their customers.

### Amazon Web Services (AWS)

We reported this issue to the AWS-Security team. They confirmed the vulnerabilities on CloudFront. The AWS-Security team stopped caching error pages with the status code `400 Bad Request` by default. However, they took over three months to fix our CPDoS reportings. Unfortunately, the overall disclosure process was characterized by a one-way communication. We periodically asked for the current state, without getting much information back from the AWS-Security team. They never contacted us to keep us up to date with the current process. For example, we only got noticed about the changed default caching policy by checking back the revision history of their respective documentation hosted in Github. Thus, we do not have much information on the noticeable amount of time required to resolve our reported CPDoS vulnerability, although having asked for it explicitly. We can only assume that this delay has to do with the large number of affected users they had to test after implementing according countermeasures. Moreover, Amazon suggests users to deploy an AWS WAF in front of the corresponding CloudFront instance. AWS WAF allows defining rules which drop malicious requests before they reach the origin server.

## Microsoft

Microsoft was able to reproduce the reported issues and published an update to mitigate this vulnerability. They assigned this case to CVE-2019-0941 [NAT19] which is published in June 2019.

## Play 1

The developers of the Play 1 confirmed the reported issues and provided a security patch which limits the impact of the `X-HTTP-Method-Override` header [Cha19]. The security patch is included in the versions 1.5.3 and 1.4.6. Older versions are not maintained by this security patch. Web applications which use older versions of Play 1 therefore should update to the newest versions in order to mitigate CPDoS attacks.

## Flask

We reported the HMO attack to the developer team of Flask multiple times. Unfortunately, we have not received any answer from them so far and hence we have to assume, that Flask-based web applications are still vulnerable to CPDoS.

## 6.7 Discussion

Using malformed requests to damage web applications is a well-known threat. Request header size limits and blocking meta characters are therefore vital means of protection to avoid known cache poisoning attacks as well as other DoS attacks such as request header buffer flow [NAT10] and ReDoS [SP18]. Also, many security guidelines such as the documentation of Apache HTTPD [Apa19], OWASP [OWA17], and the HTTP standard [FR14b] recommend to block oversized headers and meta characters in headers. CPDoS attacks, however, aims to beat these security mechanisms with their own weapons. HHO and HMC CPDoS attacks intentionally send a request with an oversized header or harmful meta character with the intent to get blocked by an error page which will be cached. Along these lines, it is interesting to see that CDN services, which claim to be an effective measure to defeat DoS and especially DDoS attacks, desperately fail when it comes to CPDoS.

According to our experiment results, most of the presented attack vectors are only feasible when CloudFront is deployed as the underlying CDN, since it is the only analyzed cache which illicitly stores the error code `400 Bad Request`. Such a non-conformance is the main reason for the HHO and HMC attacks. The other major issue for both attacks is fact that the cache forwards oversized headers and requests with harmful meta characters. Violations of the HTTP standard and implementation issues are also the main reason for many other cache-related vulnerabilities including request smuggling, host of troubles, response splitting, and web deception attacks. The HMO CPDoS attack is, however, a vulnerability which does not exploit any implementation issues and violations of the HTTP standard. The `X-HTTP-Method-Override` header or similar headers are legitimate auxiliaries to tunnel HTTP methods which are not supported by WAFs or web browsers. Play 1 and Flask returns the error code `404 Not Found` or `405 Method Not Allowed` when an unsupported action in `X-HTTP-Method-Override`

header is received. Both error messages are allowed to be cached according to RFC 7231. Akamai, CDN77, Fastly, Cloudflare, CloudFront, and Varnish follow this policy and cache such error codes. If these web caching systems are used in combination with one of the mentioned web frameworks, these combinations have an actual risk of falling victim to CDPoS attacks, even though they are in conformance with the HTTP standard and do not have any implementation issues. Therefore, the HMO CPDoS attack can be considered as a new kind of cache poisoning attack which does not exploit any implementation issues or RFC violations. This shows that CPDoS attacks do not always result from programming mistakes or unintentional violations of specification policies, but can also be the exploit of the conflict between two legitimate concepts. In case of HMO CPDoS attacks, this conflict refers to the usage of method overriding headers and the caching of allowed error messages.

Even though we did not detect attack vectors in other web caching systems and HTTP implementations, this does not mean that other constellations are not vulnerable to CPDoS attacks. As shown by Table 6.1 eight of fifteen tested web caching systems do store error pages and some of them even cache error pages which are not allowed. If an attacker is able to initiate other error pages or even cacheable error code at the target URL, then she may affect other web caching systems and HTTP implementations with CPDoS attacks as well. James Kettle, for instance, discovered two other forms of CPDoS attacks which fortunately are only successful due to specific implementation issues of the corresponding web application. The first CPDoS attack utilized the `X-Forwarded-Port` header [Ket18b]. This header usually informs the endpoint about the port that the client uses to connect to the intermediate system, which operates in front of the origin server. In the revealed attack, the cached response contained the redirect. A DoS was caused by the user's browser trying to follow the cached redirect and timing out. The second attack was able to create a DoS at `www.tesla.com` due to a faulty WAF configuration [Ket18a]. Tesla configured their WAF to block certain strings which have been used by other cache poison attacks. Unfortunately, requests with such strings were blocked by a `403 Forbidden` error page which was also cached. This shows that HMO, HHO, and HMC are not the only variations of CPDoS attacks. There are, certainly, many other ways to provoke an error page on the origin server. To the best of our knowledge and according to our experiences in developing web applications, it is not unlikely to provoke an `500 Internal Server Error` status code or other `5xx` errors in real world web applications and services. Akamai and Cloudflare do cache `5xx` error codes. At this point, we did not find a way to provoke such error messages in our experiments.

Moreover, we need to consider that contemporary web applications and distributed systems in particular are usually layered. That is, they often utilize other intermediate components such as load balancer, WAFs or other security gateways which are located between cache and endpoint. Such middleboxes or middleware may provide other request header size limits, meta character handling or header overriding features. Such systems may also react to malicious requests with error codes that could be cached.

## 6.8 Countermeasures

The most intuitive, as well as effective countermeasure, against CPDoS attacks is to exclude error pages from being cached. However, content providers which exclude cacheable error codes such as `404 Not Found` from being stored, need to consider that this setting may impair the performance and scalability. There two ways to exclude error pages from being



cached. The first approach is to configure the web caching systems to omit the storage of error responses. Akamai, CDN77, CloudFront, CloudFront, Fastly, and Varnish provide options to do so. Content providers can also add the `no-store` directive to the `Cache-Control` response header which prohibits all caches from storing the content. According to our own evaluation, all tested web caching systems except CloudFront honored the keyword `no-store` in error pages and still do so. At the time of our experiments in February 2019, CloudFront cached error pages for five minutes by default and even did so when `no-store` was included in the error response header. The only way to avoid storing error pages in CloudFront was to disable each error code from caching via the CDN's configuration interface. Fortunately, AWS changed the behavior of caching error pages after our CPDoS reporting. One important change is that `400 Bad Request` error pages are not cached by default anymore. CloudFront only caches `400 Bad Request` error messages if they include a `max-age` or `s-maxage` control directive [Ama19a].

As mentioned before, the disobey of the HTTP standard in terms of ignoring control directives is the main cause for many cache-related vulnerabilities. Beside the consideration of cache-related control directives, web caching systems must, therefore, only store error codes which are permitted by the HTTP standard. Status codes such as `400 Bad Request` are not allowed to be cached, since this error message is only dedicated to a request which is malformed or invalid. Other error codes such as `404 Not Found`, `405 Method Not Allowed` or `410 Gone` can be cached, since they provide error information which is valid for all clients. Also, HTTP implementations have to use the appropriate status code for the corresponding error case. Table 6.3 shows that almost all tested system return the status code `400 Bad Request` for an oversized request header. IIS even replies with status the cacheable `404 Not Found` error code when a limit for a specific request header is exceeded. Both error messages are not the appropriate one for requests exceeding the header size limit. According to HTTP standard, the appropriate error code is `431 Request Header Fields Too Large`. Such error information is not stored and reused by any of the tested web caching systems. To test the compliance and behavior of caches, we recommend to use the cache testing tool of Nguyen et al. [NLF19a] or Mark Nottingham [Not19].

Another very effective countermeasure against CPDoS attacks is the usage of WAFs. Many CDNs provide the option to enable WAFs in order to protect web applications against malicious requests. To avoid CPDoS attacks, content providers can configure the WAF to explicitly block oversized requests, requests with meta characters or malicious headers. Using WAFs is, however, only effective if the WAF is implemented in the cache or in front of the cache, so that harmful requests can be eliminated before they are forwarded to the origin server. The experiments in Section 6.5 and the CPDoS attack of James Kettle on `www.tesla.com` [Ket18a] show that WAFs which are integrated at the origin server such as ModSecurity do not help against CPDoS attacks. Requests which are blocked by a WAFs at the origin can still trigger an error page that is stored by the cache.

Moreover, we recommend adding a subsection to the "Security Considerations" section of the RFC 7230 [FR14b] to discuss the consequences of non-compliance with the protocol specification in order to avoid HHO, HMC and other web cache poisoning attacks. The "Security Considerations" section of RFC 7230 mentions cache-poisoning attacks including response splitting and request smuggling. However, the standard only makes recommendations that relate to these two specific attacks. The specification does not mention that the source of many cache-related attacks lies in violations of the standard. Such an additional description would increase

developers' awareness of compliance with the specifications. HMO attacks, on the other hand, cannot be avoided by complying with the standard, as they are based on non-standard means which is the `X-HTTP-Method-Override` header in this case. To avoid HMO attacks while maintaining the scalability, content providers do not need to exclude the `404 Not Found` and `405 Method Not Allowed` error code from caching. Here, vulnerable web frameworks must follow the approach of Symfony, Lavarel as well as the plugins of Django and Express.js. These HTTP implementations support the method overriding headers, but only consider to change the action when the method in the request line is `POST`. By this, a `404 Not Found` error page cannot be triggered by malicious `GET` request, since method overriding headers are ignored. When trying to poison the cache with a `POST` request with a method override header including `GET`, the returning response is not stored by any tested cache. Also, the use of non-standard headers is a general approach to conduct other cache-poisoning attacks as described by James Kettle [Ket18c]. It is the responsibility of HTTP implementations to carefully integrate non-standard headers to avoid such attacks. To analyze impact of standardized or non-standard headers in respect to caches, developers and software testers can use, e.g., the testing tools of Nguyen et al. [NLF19a] and Mark Nottingham [Not19].

## 6.9 Conclusion and Outlook

Vulnerabilities stemming from the semantic gap result in serious security threats. Distributed systems are especially prone to such attacks as they are composed by distinct layers. Their existence is one major prerequisite for the different interpretation of an object, in this case the application messages floating through the intermediaries.

In this paper we extended the known vulnerabilities rooted in a semantic gap by introducing a class of new attacks, "Cache-Poisoned Denial-of-Service (CPDoS)". We systematically study how to provoke errors during request processing on an origin server and the case, in which error responses get stored and distributed by caching systems. We introduce three concrete CPDoS attack variations that are caused by the inconsistent treatment of the HTTP method override header, header size limits and the parsing of meta characters. We show the practical relevance by identifying the amount of available web caching systems that are vulnerable to CPDoS. The consequences can be severe as one simple request is sufficient to paralyze a victim website within a large geographical region (see Figure 6.8 in Appendix B). Depending on the resource that is being blocked by an error page, the web page or web service can be disabled piecemeal (see Figure 6.6 in Appendix A).

According to our experiments 11% of the DoD web sites, 30% of the Alexa Top 500 websites and 16% of the URLs in the analyzed HTTP Archive data set are potentially vulnerable to CPDoS attacks. These cached contents include also mission-critical firmware and update files. Considering the fact that modern distributed applications often follow the Microservices [New15] and Service-Oriented Architecture (SOA) [Erl07] design principles where services are implemented with different programming languages and are operated by distinct entities, more semantic gap vulnerabilities may appear in the future. Hence, a more in-depth understanding of such vulnerabilities needs to be gathered in order to develop robust safeguards that do not depend on particular implementation and concatenation of system layers.

## **Acknowledgment**

First of all, we would like to thank all reviewers for their thoughtful remarks and comments. Moreover, we would especially like to thank Shuo Chen and James Kettle for their feedback and suggestions. Finally, we appreciated the disclosure processes with the AWS-Security team, the Microsoft Security Response Center and the Play Framework development team.

This work has been funded by the German Federal Ministry of Education and Research within the funding program "Forschung an Fachhochschulen" (contract no. 13FH016IX6).

# Appendix A: Illustrative Examples of CPDoS Attack

## A.1 Ethereum-website

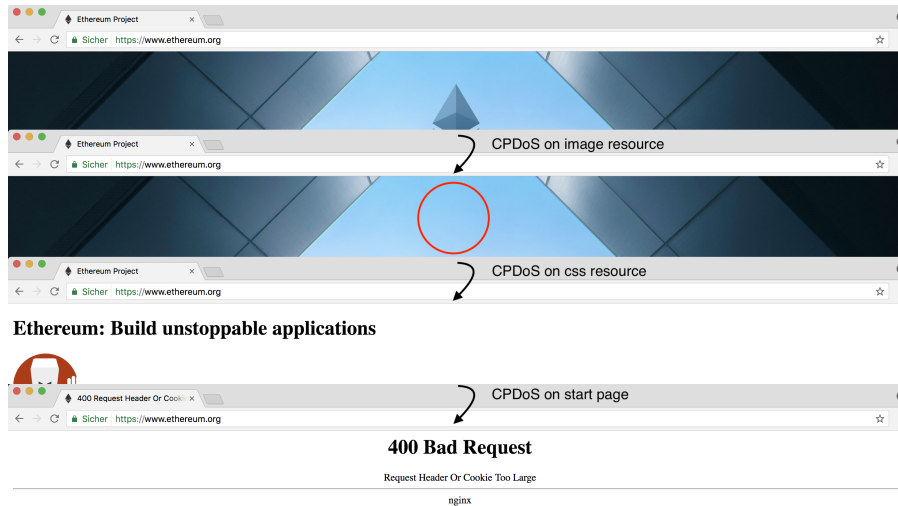


Figure 6.6: These screenshots show the start page of the website ethereum.org and how parts as well as the whole page are rendered inaccessible due to a successful CPDoS attack. More specifically, this website has been vulnerable to HHO CPDoS.

## A.2 Marines-website

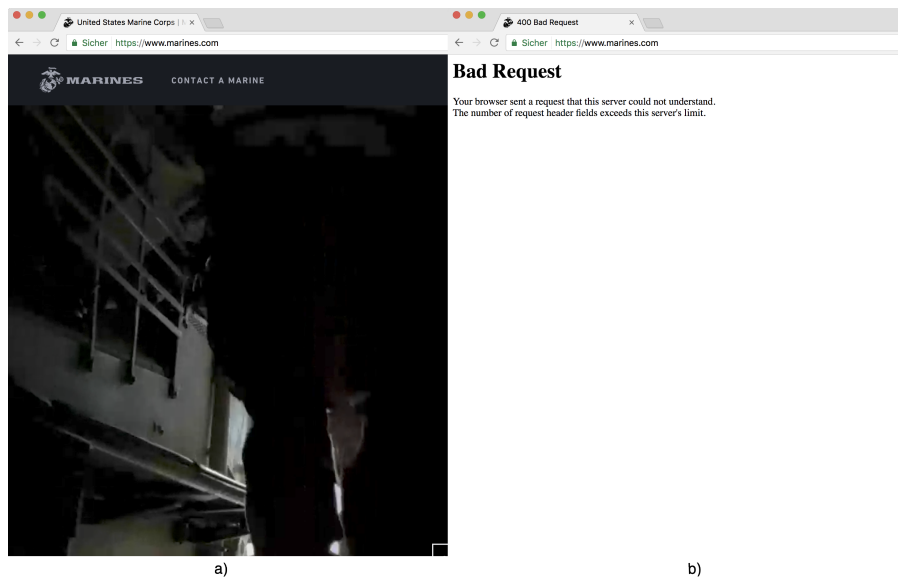




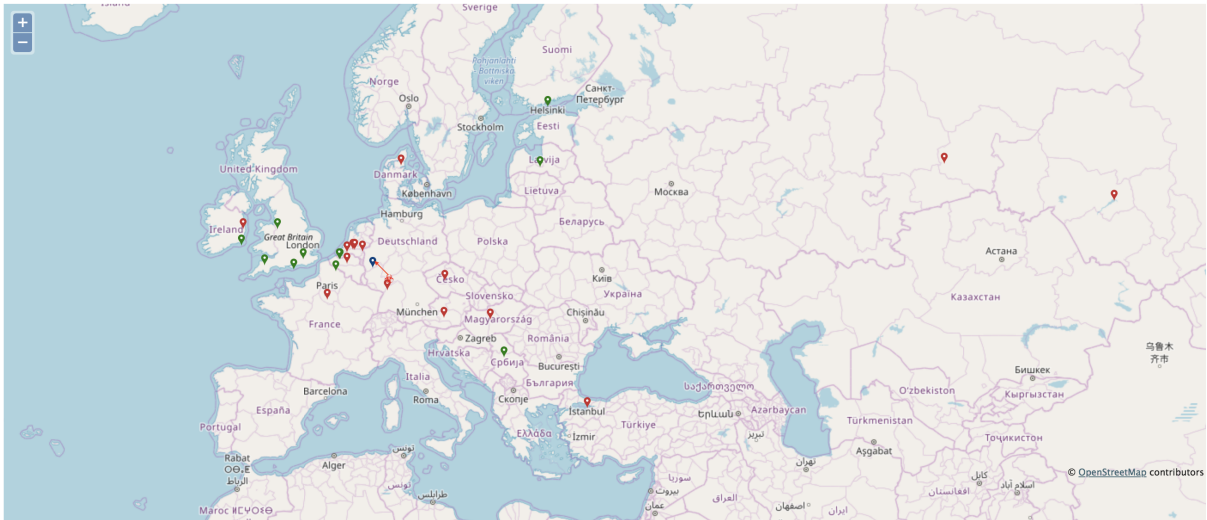


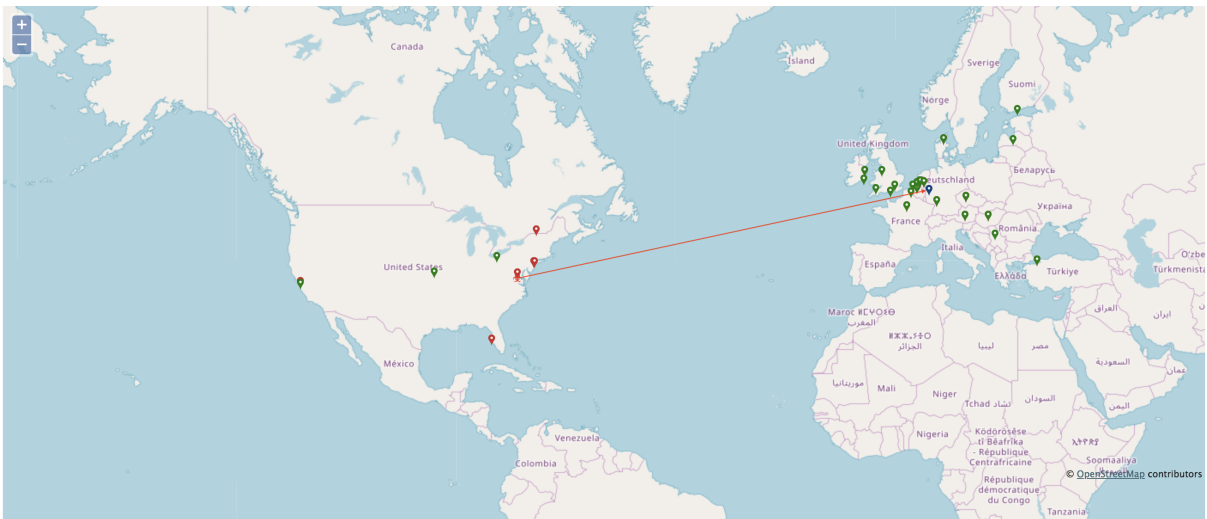
Figure 6.7: These two screenshots show the start page of the website marines.com before a) and after b) a successful CPDoS attack. More specifically, this website has been vulnerable to HHO CPDoS.

## Appendix B: CPDoS Attack Spread

Legend:  none-affected region,  affected region,  attacker,  origin server



(a)



(b)

Figure 6.8: Affected CDN regions when sending a CPDoS attack from a) Frankfurt, Germany and b) Northern Virginia, USA to a victim origin server in Cologne, Germany.

# Chapter 7

## CREHMA: Cache-aware REST-ful HTTP Message Authentication

### Summary of this publication

**Citation** H. V. Nguyen and L. Lo Iacono. *CREHMA: Cache-aware REST-ful Authentication Scheme*. In: *10th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2020. URL: <https://doi.org/10.1145/3374664.3375750>

**Status of Paper** Published

**Type of Paper** Research Paper (Conference)

**Ranking** LiveSHINE: B, Microsoft Academics: C

**Aim** In this paper, we introduce CREHMA, a REST-ful HTTP message signature scheme that guarantees the integrity and authenticity of web assets from end-to-end while simultaneously allowing service providers to enjoy the benefits of web caches.

**Methodology** Based on the knowledge gained from our studies on web caching in Paper 3, 4, and 5 (Chapter 4, 5, and 6), we extend the threat model as well as the list on to be protected REST message assets defined in Paper 1 (Chapter 2). With this background, we revisit the current state of REST-based authentication schemes and define requirements for a cache-aware REST message authentication scheme.

**Contribution** Besides the introduction of CREHMA, we study the performance, compatibility, and security of CREHMA. We found that CREHMA only introduces marginal impacts on metrics such as latency and data expansion while providing integrity and authenticity protection from end-to-end. Moreover, we show that CREHMA is compatible with existing web caching systems. In terms of security, CREHMA outperforms available REST-based authentication schemes.

**Co-authors' contribution** See Paper 6 in Section 1.1.1.

## 7.1 Introduction

In the last four decades, software has moved from simple single-user offline computer programs to complex Internet-connected distributed systems used by a multitude of people [Boo18]. Emerging trends including Cyber-Physical Systems (CSP) [Raj+10] and the Internet of Things (IoT) [Car+18] continue to foster this development. Due to the large number of users and the resulting significance for society, scalability and security are considered as two key qualities of modern distributed software [Car+18].

To ensure scalability, distributed software systems are often designed following the rules defined by the architectural style REST (Representational State Transfer) [Fie00; KJ10; BW16]. In order to scale, a system needs to be *stateless*, *cacheable* and *layered* according to REST. Stateless protocols are characterized by self-contained messages retaining the server to maintain state. Such messages are well-suited to be stored and redistributed by caches which are part of layered architectures located on the path between the client and the server. Appropriate technologies to implement REST-based architectural designs can mainly be found in the Web domain with HTTP [FR14b] at its core and related standards on caching [FNR14]. The Web is also a vivid proof that systems do scale at large when considering the REST architectural constraints in their design.

To ensure security, Transport Layer Security (TLS) [Res18] is the industry standard for protecting the confidentiality, integrity and authenticity of data in transit on the Web [Fel+17]. If the end user's client connects directly to a service provider's server then TLS is sufficient to authenticate the content served over the protected connection. However, when there is an intermediary such as a Web caching system between the end user and the service provider, the TLS connection terminates at the intermediate system's server as it requires to get access to the HTTP message to fulfill its duties [LNG19]. In this circumstance, TLS ensures that the connection to the cache server is authenticated, but it says nothing about whether the cache is serving the service provider's intended content. Essentially, both parties must completely trust the cache to faithfully serve the service's assets to its users. This includes also the proper handling of the TLS private key that the service provider needs to hand over to the cache for TLS channel termination.

Considering the fact that intermediate systems are almost ubiquitously present in Web-based applications, the required trust in such intermediaries can often not be accepted by service providers. Moreover, for critical web applications requiring a defense-in-depth in which the security is ensured by multiple safeguards, the adoption of TLS alone is by far not sufficient [Cal+19]. As an initial step to ensure end-to-end integrity protection, many so-called HTTP signature schemes have been proposed over the last years. These security schemes generate a digital signature over the HTTP message header and body in order to protect it against man-in-the-middle threats including malicious modifications and replay attacks. However, the available HTTP signature schemes either do not provide comprehensive security as they leave response messages out of the protection scope while enabling caching or they do provide signed request and response messages while preventing caching.

This current situation leads to a trade-off between using the first mentioned group of HTTP signature schemes with the abandonment of a comprehensive end-to-end security or using the second one without the advantages of caching. Many real world applications currently have to cope with this trade-off. In particular this includes organizations such as video streaming services or social media platforms, which requires an intensive usage of Content Distribution Networks (CDN) in order to delivery high quality content. Beside public resources, these organizations



also provide content, which is restricted to a group of users or a single user. In many cases the data transfer of these services is only protected by TLS. No additional end-to-end security mechanism is usually utilized to protect the exchange of credentials and cached content.

In this paper we introduce a cache-aware REST-ful HTTP message authentication scheme (abbreviated as CREHMA) that enables caches to store, modify, and reuse HTTP messages while preserving end-to-end integrity and authenticity. We make the following main contributions:

1. We **motivate the need** for end-to-end security mechanisms complementing TLS by introducing an evolved threat model for the Web that explicitly considers intermediate systems such as caches.
2. We point out that existing **HTTP signature systems have deficiencies** either in terms of inadequate protection or lack of performance without requiring modifications to caches.
3. We **introduce CREHMA**, a cache-aware HTTP signature scheme that protects cacheable resources from end to end.
4. We **evaluate CREHMA** for its compatibility with existing caches, its performance and security.

The remainder of the paper is organized as follows. In Section 7.2 we provide the required background on Web caching. In Section 7.3 we introduce an evolved threat model for Web-based systems and argue that end-to-end message-oriented security is required as complement to TLS. We present the available HTTP signature schemes in Section 7.4 and review them in the light of the threat model. We conclude with a set of requirements for cache-aware HTTP signature schemes in Section 7.5 and introduce our proposed cache-enabled HTTP signature scheme CREHMA in Section 7.6. In Section 7.7 we provide and discuss our results obtained from various experimental evaluations and conclude in Section 7.8.

## 7.2 Web Caching Background

The basic idea behind web caching is to store and reuse HTTP response messages with the aim to eliminate interactions, improving efficiency, scalability, and user-perceived performance [Fie00]. The HTTP caching standard [FNR14] distinguishes between two types of web caching systems: *private* and *shared* caches. Private caches store and reuse content of one single user, as do Web browser caches for instance. Shared caches deliver cached responses to multiple clients. Common examples include proxy caches and CDNs.

The service provider controls the cacheability of response messages by specifying its freshness. One approach is to assign an explicit *freshness lifetime* with the `max-age` directive of the `Cache-Control` response header (see Figure 7.1). By specifying `max-age=60`, e.g., the cache is instructed that the response is fresh for the next 60 seconds. During this time, the cache can store and reuse the response to satisfy recurring requests without any further interaction with the origin server. The `max-age` directive is dedicated for shared and private caches whereas `s-maxage` addresses shared caches only. An alternative approach to specify an explicit freshness lifetime for shared and private caches is to define an expiration date with the `Expires` header. If a response message includes a `Expires` header as well as a freshness lifetime defined by `max-age` or `s-maxage` directives, then the latter two must be preferred.

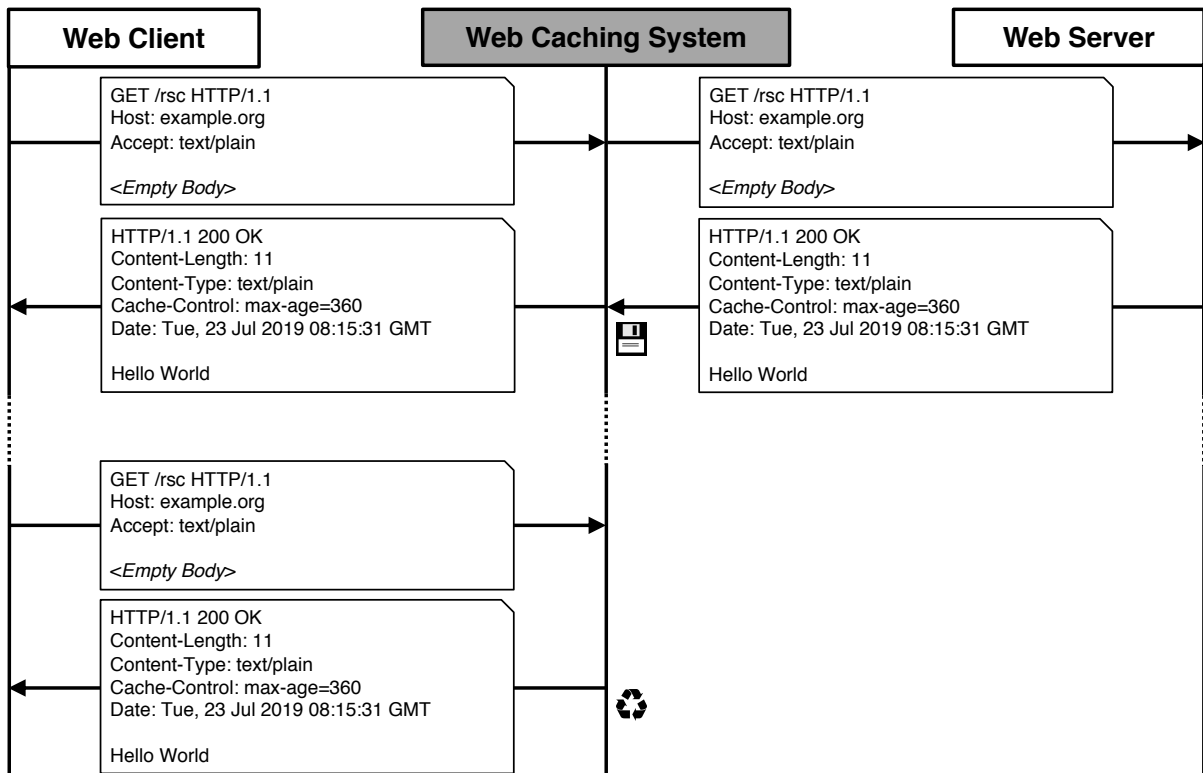


Figure 7.1: This example HTTP messages flow illustrates caching with a freshness lifetime. The response message from the origin server contains meta information declaring the amount of time the response can be reused by the cache without any further server intervention.

The second approach determines the freshness of a HTTP response by checking back with the origin server using a *conditional request* (see Figure 7.2). Service providers can explicitly force caches to validate the freshness for each incoming request. To do so, the `no-cache` directive of the `Cache-Control` header is set. This control directive tells the cache that reusing the corresponding response requires a validation of freshness with the origin server.

To aid the cache in conducting conditional requests, service providers include an opaque or time-variant validation token to the response header by which the origin server can identify whether the cache owns a fresh copy of the response. The example in Figure 7.2 shows a freshness validation with an opaque validation token that is placed in the `ETag` response header. Time-variant validation tokens are respectively included in the `Last-Modified` response header. When the cache constructs a conditional request, the opaque token is placed in the `If-None-Match` request and the time-variant token is added to the `If-Modified-Since` header. If a conditional request contains the `If-None-Match` and the `If-Modified-Since` headers simultaneously, then origin server has to prefer the `If-None-Match` header. If the conditional request is successful, i.e., the stored response is still fresh, the origin server sends a response with the status code `304 Not Modified` and an empty body. The cache is now allowed to reuse this response message, but it has to update the headers of the stored response with the ones contained in the server's response. Such a header update is emphasized in Figure 7.2. Here, the cache updates the `Date` header of the stored response with the one of the received `304 Not Modified` response. If the origin server notices that the validation token refers to a stale response message, it returns a full response message including a new header and body. The

cache must forward this new response message to the client. Moreover, the cache must replace the new response message with the stored one.

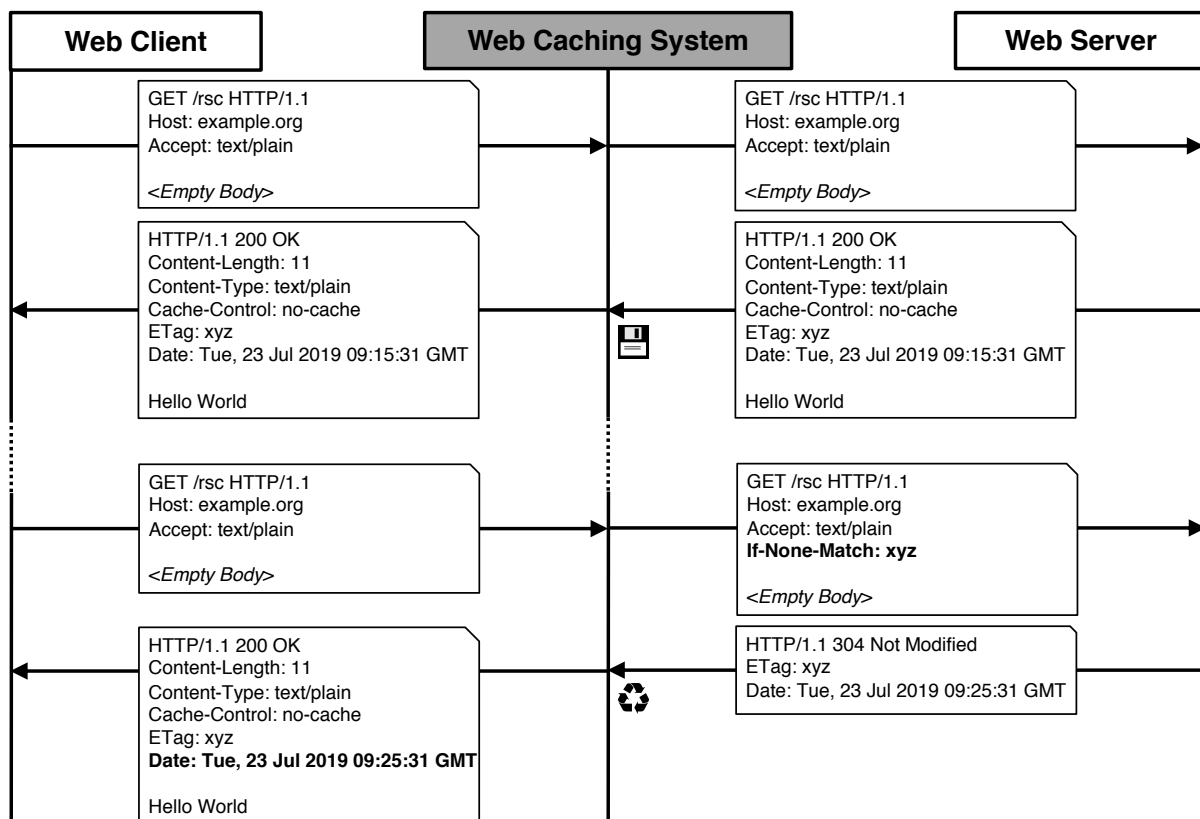


Figure 7.2: This example HTTP messages flow illustrates caching with freshness validation. The response message from the origin server contains meta information instructing the cache to validate the resource’s freshness with the origin server before reuse. The bold entries denote headers that are added or updated by the cache.

A service provider can also explicitly restrict caches from storing certain responses with the `no-store` directive of the `Cache-Control` header. Another relevant control directive of the `Cache-Control` header is `no-transform`, which prohibits any intermediate system to change the body of traversing and cached messages.

When a response lacks any caching directive, caches can derive the freshness implicitly by themselves. As with explicit caching, a cache can ensure the response freshness with two approaches. It can define a freshness lifetime by using heuristics or if can validate the freshness with a conditional request. Implicit caching is optional. Caches can also omit such response messages from caching.

### 7.3 Evolved Threat Model

To ensure end-to-end security in modern Web-based distributed software systems, the existence of intermediate systems must be taken into account. Instead, the current threat model for the Web, and TLS in particular, still assumes that a client is directly connected to a server. If this assumption is true, then TLS is sufficient to secure the content provided over the connection. However, as soon as an intermediary, including a cache, joins this setup, the TLS connection

terminates at the cache. In this case, the TLS connection provides its security services between the client and the cache, but it says nothing about whether the cache is serving the provider’s intended content. Therefore, the service provider and the end user must fully trust the cache to faithfully serve the web service’s resources. Until today they have no other choice but to assume that caches are acting accordingly.

Service providers and end users should be alarmed about this critical prerequisite, since intermediate systems and caches in particular are more and more deployed as third-party services. Simply trusting these parties is often not sufficient, especially when considering the steady growth of P2P-based caching systems [Lov17]. Moreover, the threats and resulting consequences when caches do not act faithfully can be severe. Caches can for instance (A1) modify web service’s content or metadata to distribute false information or deface the victim, (A2) serve stale content to prevent software updates to take effect, (A3) inject malicious client-side code to eavesdrop sensitive information including credentials, and (A4) responding in arbitrarily malicious ways to client requests.

The motivation for caches to do this can be manifold and can even include powerful organizations or governments forcing them. Therefore, the current threat model needs to evolve with the architectural changes of Web-based distributed systems. In general, an enhanced threat model should provide end-to-end confidentiality, integrity and authenticity for data on the Web in the presence of intermediaries. Such enhanced security schemes need to be understood as complementary to TLS to enable defence-in-depth.

In recent years, many HTTP signature schemes have been proposed that provide end-to-end integrity and authenticity. As per the status quo, these schemes have never been analyzed according to their interdependencies with caches.

## 7.4 Review of HTTP Signature Schemes

	Request Protection	Response Protection	Response Swapping Protection	Replay Attack Protection	Cache Compatibility	Allow Benign Message Modification
AWS [Ama19b]	●	○	○	●	●	●
Google [Goo17]	●	○	○	●	●	●
HP [Hew14]	●	○	○	●	●	●
Microsoft [Mic17]	●	○	○	●	●	●
SHREQ [Run19]	●	○	○	●	●	●
OAuth1 [Har12]	●	○	○	●	●	●
OAuth Mac Tokens [RMT14]	●	○	○	●	●	●
Signing an HTTP Request for OAuth [RBT16]	●	○	○	○	●	●
Lo Iacono et al. [LNG19]	●	●	○	●	○	●
Cavage et al. [CS19]	●	●	○	●	○	●
Serme et al. [Ser+12]	●	●	○	●	○	○
SRI [Dev+16]	○	●	○	○	●	●
Stickler [LCB16]	○	●	○	●	●	●

Table 7.1: Analysis of the related work in HTTP signature schemes

The goal of HTTP signature schemes is to provide end-to-end integrity and authenticity by digitally signing the HTTP messages or certain parts of them. Although the available schemes are different from each other, they still follow a common procedure when signing and respectively verifying HTTP messages (see Figure 7.3).

The signing procedure starts with processing the HTTP message, which is to be signed. First, a string representation is generated from the message by concatenating a time-variant parameter (TVP), the entries of the message headers, and the hash of the message body. This constructed string is then signed with a signature generation key. After that, the `Signature` header is built

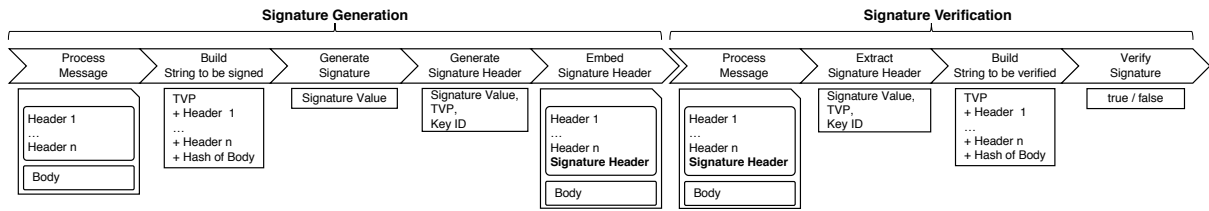


Figure 7.3: Process diagram visualizing the general steps all HTTP signature schemes adopt when signing and verifying HTTP messages

by concatenating the string representation of the key ID, the TVP and the signature value. This new header is finally added to the HTTP message headers as emphasized in Figure 7.3.

Accordingly, the verification procedure starts with processing the signed HTTP message. First, the `Signature` header is extracted from the message and the contained items are retrieved from it. In the following step, the string to be verified is constructed by concatenating the string representations of the TVP, the entries of the message headers, and the hash of the body. The last step verifies the signature and checks the validity of the TVP in order to detect replay attacks.

As said before, the available HTTP signature schemes follow this general process with slight deviations. Amazon Web Services (AWS), Google, Hewlett Packard (HP), and Microsoft use self-developed HTTP signature schemes for users to authenticate to their cloud storage services [Ama19b; Goo17; Mic17; Hew14]. Their schemes do make use of a TVP to avoid replay attacks, but they do neither sign the response nor all request headers. Instead, they define a set of request headers that is considered security-relevant, including the HTTP method, the URL and the `Host` header. These headers have to be non-empty and present exactly once for the message to be valid. The computed signature value and the signature-related metadata are then placed in the `Authorization` header, which is finally added to the request.

Signed HTTP Requests (SHREQ) [Run19] is very similar to the industry-driven approaches, as it provides end-to-end integrity for request messages only. It signs a subset of request headers and adds a TVP to the signature to avoid replay attacks. The signature value and the corresponding metadata are either included in the body or in case of requests without a body in the URL.

The first version of the authorization framework *OAuth* [Har12] as well as the *OAuth 2.0* draft extensions *OAuth Mac Tokens* [RMT14] and *Signing an HTTP Request for OAuth* [RBT16] also sign requests only following a predefined list of to be protected headers. While *OAuth 1.0* has been established as an official web standard, the both draft specifications have not been updated since 2014 and 2017.

Cavage et al. [CS19], Lo Iacono et al. [LNG19], and Serme et al. [Ser+12] proposed HTTP signature schemes that protect the request as well as the response message. The scheme by Serme et al. includes a TVP in the signature to detect replay attacks and embeds the signature value as well as the signature metadata in signature headers starting with `X-JAG-*`. However, it does not define a list of to be signed headers. Instead, all headers are considered in the signature and verification procedure. This may impair the signature's validity when an intermediate system modifies or adds a protected header. Caches usually add a header to traversing request messages to validate the response freshness (see Figure 7.2). If this happens, the string to be signed built by the client and the string to be verified crafted by the origin server are different. This issue hampers web applications to use the security mechanism of Serme et al. in conjunction with caches. Cavage et al. and Lo Iacono et al. instead provide a list of security-relevant message

headers. Headers not part of the list can be modified and added by intermediaries. Both signature schemes also require to integrate a TVP to avoid replay attacks. The resulting signature value and the corresponding metadata is inserted in the introduced `Signature` header. A major issue that affects all three approaches is a missing distinction between a replay attack and benign replayed message. A reused cached message contains the same signature value and TVP as the original one. Clients receiving a reused signed response message twice, may assess this signed replication as a replay attack, which is the case for all three HTTP signature schemes as they are not compatible with caches. As the current state does not provide a solution for this issue, web applications which consider to use these security mechanisms are forced to disable caching. The other HTTP signature schemes which only protect the request message can be used in conjunction with caches. Here, the client does not have to deal with the distinction between replay attacks and reused signed response messages from a benign cache, as there is no end-to-end security and replay attack protection for the response message at all. Another drawback of all three HTTP signature schemes is the missing protection against attack vectors which swap signed responses.

Many web applications use Subresource Integrity (SRI) [Dev+16] for providing end-to-end integrity of static files. SRI is used in conjunction with CDNs to distribute public resources. The integrity of the content is ensured by a message body hash generated by the service provider. The hash value is included in an attribute of the corresponding HTML tag (e.g. `<script ... integrity='sha384-...'>`). The web browser verifies the integrity by comparing the hash value in the HTML attribute with the self-calculated hash value. This hash value only ensures the integrity of the response message body. Neither the request nor any message header is protected by SRI. A compromised or malicious cache can still replace the hash value with a hash value of a malicious content. Web browsers cannot detect such a man-in-the-middle attack, as SRI does not contain any mechanism to verify the authenticity.

Stickler [LCB16] is another end-to-end integrity protection scheme for cacheable static Web assets. The goal of Stickler is to ensure end-to-end integrity in Web-based application without the need to trust the cache for integrity. When a client visits a Stickler-protected website, the origin server returns an HTML page including a bootloader script. This script contains a public key, a script for retrieving and verifying the protected assets and the location of a manifest file. The integrity and authenticity of the bootloader script is ensured by a TLS connection. The bootloader starts with invoking the manifest file containing a manifest signature. If the signature is valid, the bootloader retrieves the website's assets based on a list included in the manifest. Each asset in this list contains the asset's URL, the corresponding hash or signature value and optionally an expiration date in order to prevent caches from serving stale or outdated content. Since the integrity and authenticity of the manifest is ensured by a signature, the client can verify the integrity and authenticity of assets by comparing the assets' hash or signature value. A shortcoming of Stickler is that it does not provide a replay attack protection for the manifest file, as it only contains a signature without an expiration date. A malicious cache is henceforth able to replace the current manifest file with an outdated one. Another major drawback of Stickler is the exclusive protection of the response body. The response headers remain unprotected. Available HTTP attacks have shown, however, that the manipulation of the response header can have severe consequences [Kle04; LNG19].

A summary of this related work review is given in Table 7.1. It gets apparent that there is a lack of comprehensive protection means for HTTP messages that are cache-enabled.



## 7.5 Requirements for Cache-aware HTTP Signature Schemes

The related work analysis in the previous section highlights that many end-to-end message signature schemes for HTTP have been proposed in recent years with some of them originating from global players such as AWS, Google, HP, and Microsoft. This emphasizes the significance as well as demand for end-to-end security schemes. However, Section 7.4 also shows the shortcomings of the available HTTP signature schemes in terms of comprehensive protection and cacheability. Our goal is to close this gap and to define a cache-aware HTTP signature scheme that aligns end-to-end integrity protection and cacheability. Based on the threat model introduced in Section 7.3, we define the requirements for a cache-aware HTTP signature schemes as follows:

- (R1) A cache-aware HTTP signature scheme must ensure the integrity and authenticity of request as well as response messages. This includes a holistic protection for the message headers and body.
- (R2) A cache-aware HTTP signature scheme must pair the request and the corresponding response message in order to alter response message swapping.
- (R3) A cache-aware HTTP signature scheme must enable caches to modify signed messages while maintaining semantic and logical equivalence.
- (R4) A cache-aware HTTP signature scheme must tell replay attacks apart from replayed messages delivered by a benign cache.

## 7.6 CREHMA

CREHMA is based on the work of Lo Iacono et al. [LNG19], as it already fulfills requirements (R1) and (R3) of Section 7.5. Moreover, the scheme provides a comprehensive and detailed description on what security-relevant elements are required to be integrity protected. Cavage et al. [CS19] fulfills both requirements as well, but it fails in protecting all security-relevant headers [LNG19] and it is still in an early development stage with many parts in the draft being still not completely specified. CREHMA extends the approach of Lo Iacono et al. with the aim to make it cache-ware.

Lo Iacono et al. denoted their scheme as REST-ful HTTP Message Authentication (REHMA). One main contribution of REHMA is a comprehensive list of mandatory HTTP request and response headers. The defined message elements are required to be available in order to render a HTTP message self-descriptive for endpoints as well as intermediate systems. Since these message elements are mandatory, they are considered as security-relevant and are therefore protected by the signature.

To enable cacheability, we adjust the REHMA policy and add cache-related response headers (see Table 7.2). As with the REHMA policy, the CREHMA header policy requires to consider the `Content-Type` header, `Content-Length`, `Transfer-Encoding` header and the HTTP version in the signature and verification process of each HTTP message. In case the message is a request message, the `Accept` header, the `Host` header, the HTTP method and the request target must be integrity protected. In most cases, the request target is the URL path including the query string. When the message is a response message, the status code



HTTP Request Message	HTTP Response Message
HTTP Method	<b>Cache Key</b>
Request Target	HTTP Version
HTTP Version	Status Code
Accept	<b>Cache-Control</b>
Content-Length	Content-Length
Content-Type	Content-Type
Host	<b>ETag</b>
Transfer-Encoding	<b>Expires</b>
	<b>Last-Modified</b>
	Transfer-Encoding
	<b>Vary</b>

Table 7.2: HTTP headers that need to be signed as defined by REHMA [LNG19]. CREHMA extends this list with the bold entries to protect against malicious caches

must be considered by the signature. Moreover, the CREHMA header policy includes the `Cache-Control`, the `Expires`, the `Last-Modified`, the `ETag`, and the `Vary` response header. These response headers are vital cache-related meta information which are required to be protected by a cache-aware HTTP signature scheme. Not protecting these headers enables a man-in-the-middle to manipulate the caching behavior. Note, CREHMA does not consider any cache-related request headers, as they are barely supported by caches [NLF19a].

Beside the addition of the mentioned to be signed headers, one major difference between REHMA and CREHMA is the consideration of the cache key in the message signature and verification process of the response message. The cache key is an essential indicator for caches to assign an incoming request to a stored response. According to the HTTP caching standard the cache key consists of the HTTP method and the URL. None of the available HTTP signature schemes takes the cache key into account. The omission of the cache key allows a man-in-the-middle to replace a response message with another response message which are signed with the same key so that the client receives a signed response which is not dedicated to the actual request message. For instance, an malicious intermediate system or an attacker which is able break the TLS connection can exploit this shortcoming to exchange a signed patched script file with a signed script file that still contains vulnerabilities. The man-in-the-middle can also, e.g., substitute signed images or videos with another signed images or videos to manipulate news or posts on social media platforms. The integration of the cache key in the signature processes allows to detect such attack vectors as the cache key of the client's expected response message is different to the one of the replaced response message. To build the cache key for the signature or verification process of the response message, the HTTP method, the request target and the `Host` header must be obtained from the received or issued request message. The cache key can also be extended by other headers such as the `User-Agent` or `Accept-Language` header. The origin server can inform the client about the cache key extension with the `Vary` header (e.g. `Vary: Accept-Language, User-Agent`). If this is the case, then these cache key extension headers the must be considered in the signature and verification process as well.

### 7.6.1 Signature Generation

An endpoint, i.e client or origin server, starts the signature generation with processing the message (see left half of Figure 7.4). CREHMA defines two different templates for building the string to be signed: one for the client's request message and one for origin server's response message. Both templates require to add a TVP firstly. If a request message is processed by the client, the HTTP method, the request target and the HTTP version are added next. These

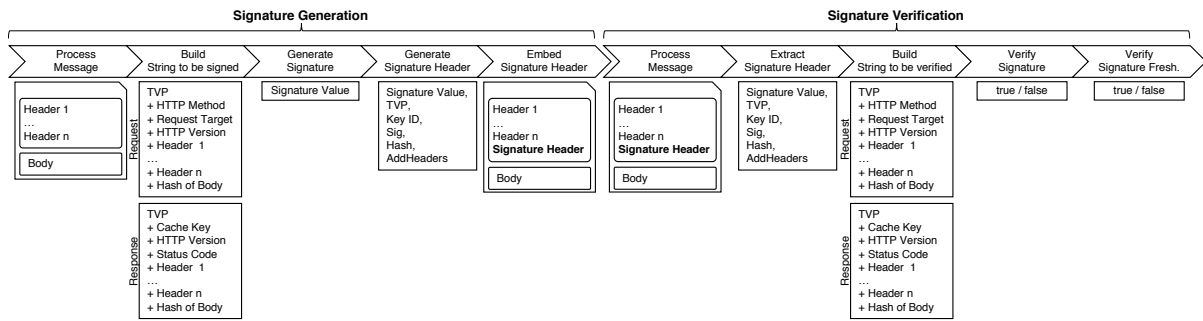


Figure 7.4: Process diagram visualizing the steps to sign and verify HTTP messages using CREHMA

header entries can be obtained from the processed request message. The origin server which intends to sign a response message concatenates the cache key, the HTTP version, and the status code to the string to be signed. The three latter header entries can be extracted from the processed response message. The cache key is obtained from the received request message. After that, all the other headers specified in the CREHMA header policy (see Table 7.2) are included according to the alphabetical order of the header names. This applies to both messages. This means, the client extracts the header entries defined in the left side of Table 7.2 from the request message and appends them alphabetically to the string to be signed. The origin server does the same but uses the right side. Note, that all headers of the CREHMA header policy must be considered in the construction of the string to be signed and verified respectively. When signing the request message, the HTTP method, the request target, the HTTP version, and `Host` header must not be empty. For the other mandatory headers, empty values are accepted. For instance, the `GET`, `HEAD`, `DELETE`, and `OPTIONS` requests contain an empty body. Therefore, it is not required for a `GET` request to include the `Content-Length`, `Content-Type`, and `Transfer-Encoding` headers. This is also true for the `Accept` header, which is only required when a client desires a distinct media type. The omission of this header indicates that the client accepts all types of media. Still, according to the CREHMA header policy these headers must be part of the string to be signed. Thus, empty string values are added as placeholders. If a response message is about to be signed the HTTP version and the status code must be available. The cache key must be obtained from the received message. All other header entries defined in the right side of Table 7.2 can be replaced with an empty string if not present. The last item to be added for both messages is a text representation of the message body's hash value. If the message body is empty, then the hash of an empty value is appended instead. The next step signs the constructed string. A text representation of the computed signature value is added to the `Signature` header, which also comprises the TVP value, the key ID as well as the names of the used signature and hash algorithms. Additionally, the `Signature` header includes an `AddHeaders` value which can be used to inform the verifying party that additional application-specific headers are protected by the signature. The final step adds the constructed `Signature` header to the message as emphasized in Figure 7.4.

## 7.6.2 Signature Verification

As with the signature generation, the client or the origin server starts the signature verification procedure with processing the message (see right half of Figure 7.4). The next step extracts the `Signature` header from the message in order to obtain the information to verify the signature

value. The `Signature` header must exist exactly once in a message. Duplicate or missing `Signature` headers result in classifying the message as invalid. This also applies for all other headers of the CREHMA header policy. After successfully extracting the `Signature` header, the string to be verified is build according to same policy and order as for the string to be signed. The client receiving a response message builds the string to be verified according to template for the response message. The to be verified header entries as well as the body can be obtained from the response message and the cache key can be retrieved from the client's issued request message. The origin server concatenates the string to be verified according to the template for the request message. With the string to be verified and the parameters of the `Signature` header, the procedure can now verify the authenticity and integrity of the received message. Besides the validation of the signature value, the verification process also checks whether the TVP value is within a given time frame. Moreover, it is required to check whether the signed message is a replay attack or a reused signed response message form a benign cache. To distinguish between a replay attack and a legitimate reused message, the verification process validates the signature freshness which is equivalent to the response freshness. The signature freshness can be either inferred from the `max-age` or `s-maxage` control directive which is included in the `Cache-Control` or it can be derived form the `Expires` header. The parameters `max-age` and `s-maxage` describe the relative explicit freshness lifetime. This information can be trusted, since the `Cache-Control` header is part of the CREHMA header policy and therefore protected by the signature. To compute the signature freshness the seconds defined in the `max-age` or `s-maxage` are added to the timestamp defined in the TVP value. The resulting timestamp is compared to the current time. If the comparison indicates that the resulting timestamp is not in the past, then the signature can be considered as fresh. If the explicit freshness lifetime is defined by an absolute expiration date with the `Expires` header, then this timestamp is used for the comparison. The `Expires` header can be trusted as well, as it is included in the CREHMA header policy.

Note, that signature freshness only exists for response messages. If the verification process detects a reused request message signature, this message must be considered as a replay attack. Also, if a signature value of a response message is received twice but the message header neither contains a `Expires` header nor a `max-age` or `s-maxage` control directive, then is message must be classified as invalid. Moreover, the signature and verification process for caching with freshness validation requires an additional step as well.

### 7.6.3 Use Cases

We present two examples use cases to further illustrate CREHMA. The use cases are crafted following the the two main caching approaches, namely caching with freshness lifetime and with freshness validation (see Section 7.2).

#### Caching with Explicit Freshness Lifetime

Let's assume the request and response message flow in the Figure 7.1 requires to be protected by CREHMA. Following the signature generation and verification policies of CREHMA and the templates for the request and response messages, the constructed strings are shown below (based on the first HTTP message shown in Figure 7.1):

String to be signed/verified for request message	String to be signed/verified for response message
2019-06-13T15:41:10.494Z GET /rsc HTTP/1.1 text/plain  example.org  47DEQpj8HBSa-_TImW-5JCeuQeRkm5...	2019-06-13T16:41:21.233Z GET example.org/rsc HTTP/1.1 200  max-age=360,no-transform 11 text/plain  pZGmlAv0IEBKARczz7exkNYsZb8LzaMr...

The string representation of the request message shows that the TVP, the HTTP method, the target resource, and the HTTP version are appended according to the order of the template in Figure 7.4. The remaining mandatory headers follow in alphabetical order. The headers that are not present are represented by an empty string value.

Figure 7.5 shows the HTTP message flow in which caching with an explicit freshness lifetime is used. The message flow is analogous to Figure 7.1 with the exception that these messages contain CREHMA signatures. For the sake of readability, we removed the `Date` headers though. All messages include the `Signature` header which contains the signature value and related metadata. As both messages do not require to integrate additional application-specific headers to the signature and verification process, the `addHeader` parameter contains the value `null`. If both endpoints intend to integrate additional security-relevant headers to the signature and verification process, a list with the header names using semicolon as delimiter must be included to the `addHeader` parameter (e.g `addHeaders=Content-Security-Policy; X-XSS-Protection`).

The signed response message contains a `Cache-Control` header with the directive `max-age=360` which indicates that this response message can be reused for the next six minutes. If a client receives a signed response twice, then the verification process needs to add the seconds of the `max-age` directive value to the timestamp of the TVP value. If this sum is greater than the current time, the reused signature is considered as fresh. Beside `max-age=360` the `Cache-Control` header also contains `no-transform`, which prevents intermediate systems from modifying the message body.

## Caching with Freshness Validation

Figure 7.6 shows an example HTTP message flow involving caching with explicit freshness validation. Although the messages have been protected using the CREHMA signature scheme, the flow and especially the HTTP messages are mostly similar to the one in Figure 7.2. Again, we omitted the the `Date` header for readability reasons.

When the response is requested subsequent times, the cache converts the request into a conditional one by adding the `If-None-Match` header and forwards the conditional request to the origin server. Since the `If-None-Match` and the `If-Modified-Since` headers are not part of the mandatory request headers, they can be added by the cache without interfering with the request's signature. If the conditional request is successfully processed by the origin server, it returns a signed response message with the status code `304 Not Modified` as shown in Figure 7.6. This response message contains only the response line and the `ETag` header with

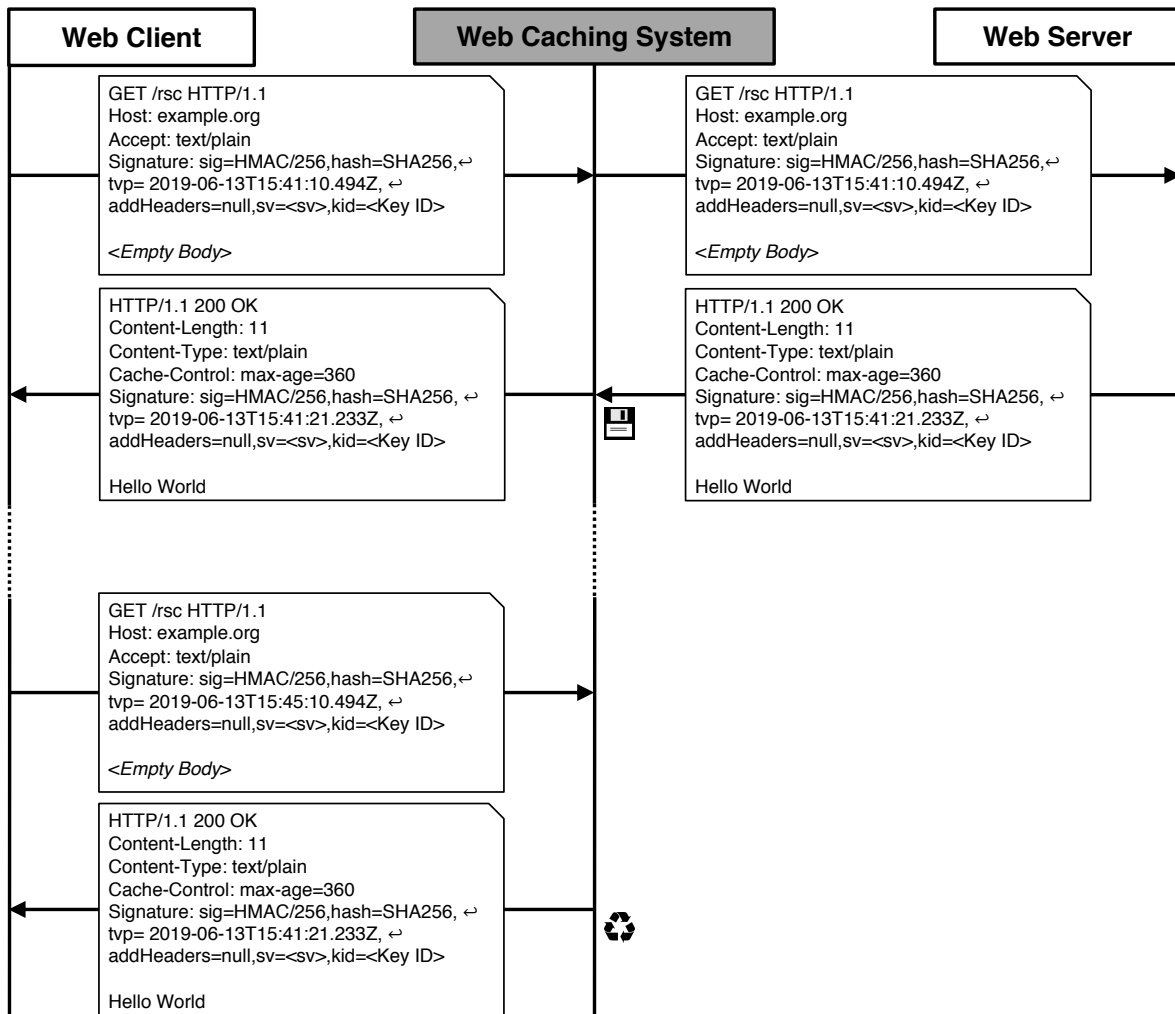


Figure 7.5: This example HTTP messages flow shows caching with a freshness lifetime. The messages are integrity protected by CREHMA.

value `xyz` indicating that the cached response message is still up-to-date. Once a cache receives such a message, it is allowed to reuse the cached response but it must update the header entries of the stored response with the ones of the `304 Not Modified` response.

CREHMA makes use of the header updating policy in order to refresh the expired signature value of the stored response message in the cache. To do so, the origin server has to compute two signatures: One to update the expired signature of the cached response and the other is required to verify the integrity and authenticity of the `304 Not Modified` response. The string to be signed of the first signature must be constructed under the assumption that the full response message with a header and the body is returned (see left column in the table below). This string is almost identical to the one which was constructed for the initial response message in Figure 7.6. The only difference is the updated TVP value. To accelerate signature generation time, this string to be signed without the TVP value should be stored in server-internal database. The database index for this string could be the `ETag` header value. When creating this signature for the next time, a new TVP value needs to be added only. The second signature must be crafted according to the message elements of the `304 Not Modified` response message as shown by the right column in the table below.

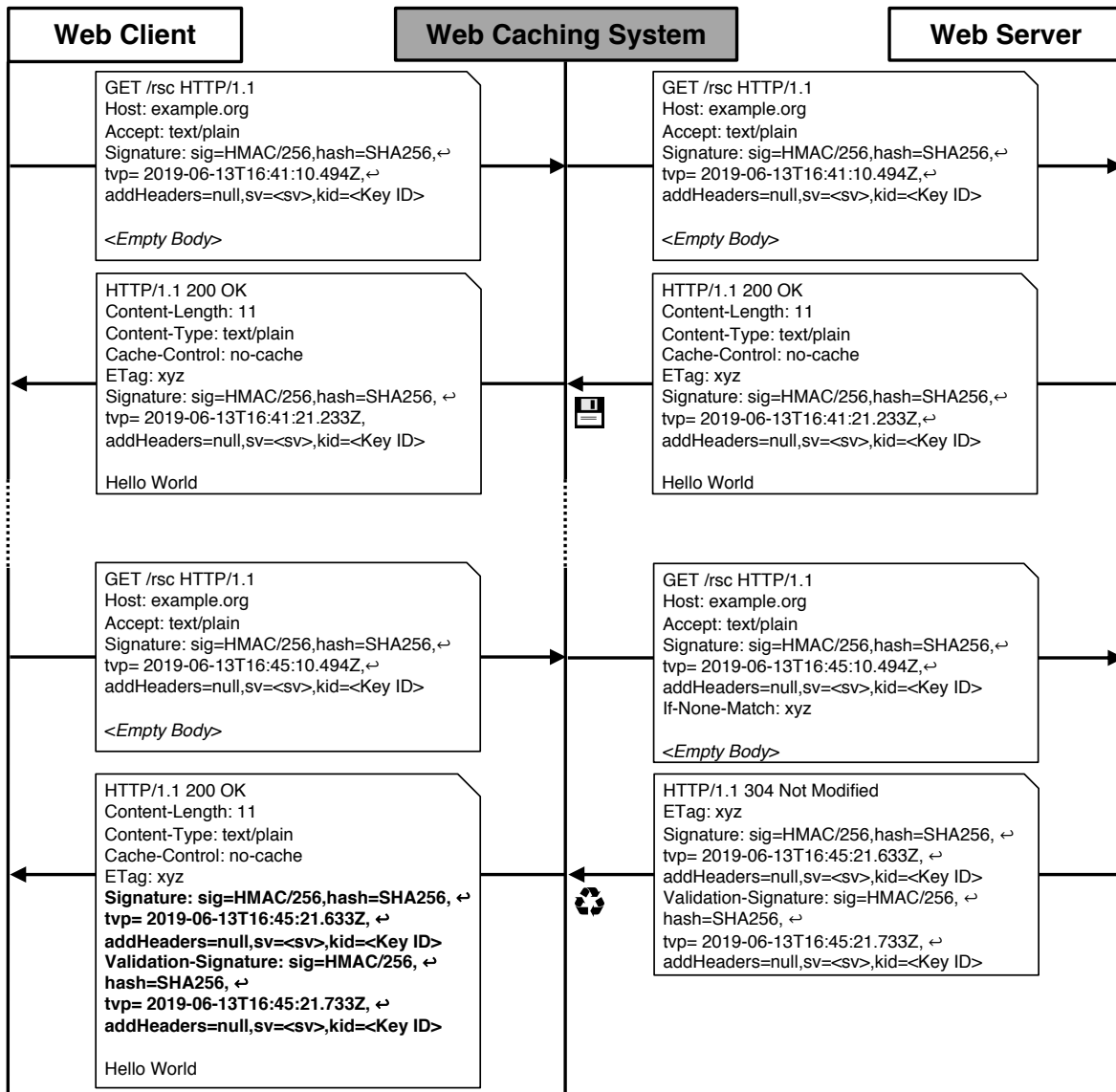


Figure 7.6: This example HTTP messages flow shows caching with freshness validation. The messages are integrity protected by CREHMA.

The signature value and the corresponding metadata which are intended to update signature value of the cached response message must be inserted the `Signature` header. The other signature value in conjunction with its meta information which protects the 304 Not Modified response message must be included the newly introduced `Validation-Signature` header as shown in Figure 7.6.

The returned 304 Not Modified response message requires the cache to update the header entries of the stored response with the headers of the 304 Not Modified response. Accordingly, the `Signature` header of the stored response message gets updated too. All updated headers are marked in bold. The response with the updated header can now be returned to client. As the `Signature` header is refreshed, the client can successfully verify the message signature. The addition of two signatures in the 304 Not Modified response message is mandatory, as the origin server does not know whether the client or what intermediate system includes the `If-None-Match` header to the request message. If client has inserted these headers by itself

string to be signed/verified of response message signature to be updated	string to be signed/verified of 304 Not Modified response message
2019-06-13T16:45:21.633Z GET example.org/rsc HTTP/1.1 200  xyz no-cache,no-transform 11 text/plain  pZGmlAv0IEBKARczz7exkNYsZb8LzaMr...	2019-06-13T16:45:21.733Z GET example.org/rsc HTTP/1.1 304  xyz  47DEQpj8HBSa-_TImW-5JCeuQeRkm5...

because it operates an own cache or the client is a caching middlebox by itself which supports CREHMA, it must validate the signature value in the `Validation-Signature` header, as this security information verifies the authenticity and integrity of the 304 Not Modified response message. If a client decides to add to the `If-None-Match` to the request message, it should not include this elements as additional to be protected header entries in the `addHeaders` parameter. This is also true for the `If-Modified-Since` header. According to the RFC 7234, caches can append further opaque validation token to the `If-None-Match` header or can change the value in the `If-Modified-Since` header. Such modifications impair the signature value of the request. In real world scenario, a client does not know how many and what intermediate systems a request message traverses until it reaches the origin server. Therefore, the `If-None-Match` and the `If-Modified-Since` header should be excluded from the signature and verification process.

#### 7.6.4 Limitations

With CREHMA, distributed web-based software systems can benefit from caching without having to trust the cache for integrity. CREHMA achieves these properties while remaining opaque for the cache. Still, CREHMA does have some limitations. One is, that CREHMA supports explicit caching only, since it requires reliable information for clients to unambiguously infer the signature freshness. Signed responses which are stored and reused according to an implicit freshness lifetime do not contain any reliable indicator for the client to validate the signature freshness. To hinder caches from deriving an implicit freshness lifetime, response messages which are not considered to be cached must be declared with `Cache-Control` header containing the control directive `no-store`.

Moreover, CREHMA does not allow intermediate systems to transform the message body. For instance, some CDNs provide the feature of minifying assets such as scripts, stylesheets and images to reduce the data volume required to be transferred. All kinds of compression and other kinds of message body modifications are not permitted, since a change of the message body changes its hash fingerprint and thus void the signature's validity. To prevent caches from modifying the response message body, the `no-transform` control directive must be added to the `Cache-Control` header. In case this control directive has not been set by the origin server, the CREHMA signature generation adds it to the respective header.



## 7.7 Evaluation

To evaluate the introduced CREHMA signature scheme, we designed and implemented a set of distinct experiments to gain insights on the compatibility of CREHMA with real world caches as well as its performance and security properties. The test beds consist of two prototype clients, a set of real world caches and a prototype test server. One client is based on Ruby v2.5.1p57 using the Net:HTTP and OpenSSL module for creating hashes as well as signatures. We used this client to evaluate proxy caches and CDNs. The other client is a single page web application implemented with Javascript and the Web Crypto API. We utilized this Javascript-based client to test web browser caches. We developed the test server based on node.js v12.4. The test server utilizes the native http and crypto node modules for computing hashes and signatures. In all our compatibility and performance evaluations, we used HMAC/SHA256 to generate the signatures and SHA256 to calculate the hashes. Moreover, we used Base64URL to encode the hash and signature values.

### 7.7.1 Compatibility

In order to meet requirement R3, we have designed CREHMA to be transparent to caching systems, i.e. in CREHMA-instrumented systems, clients and servers must be adapted while the caches remain unchanged. To validate whether our design influences the proper functioning of caches or CREHMA and downstream processes, we conducted experiments with real world caching systems. We performed our experiments with three different types of caches: web browser caches (Chrome v.75.0.3770.100, Safari v12.1.2, Firefox v69.0, Microsoft Edge v17.17134), proxy caches (Apache Traffic Server (ATS) v8.0.3, Apache HTTPD v2.4.41, nginx v1.16.1 and Squid v4.8) and CDNs (CloudFront and CloudFlare). Our compatibility evaluations covered caching scenarios with explicit freshness lifetime and explicit freshness validation.

In the test procedure to evaluate the handling of CREHMA-protected messages equipped with an explicit freshness lifetime, the client sends an initial signed GET request to the server. The server verifies the request and returns a signed response containing the `Cache-Control: max-age=3600` header instructing the cache to store and reuse the response for the next hour. After one second the client executes an equivalent request and inspects the source and validity of the response. We repeated this procedure with the `s-maxage` directive and the `Expires` header.

As shown in Table 7.3, all examined caches were able to store and reuse CREHMA-signed responses with an explicit freshness lifetime. The CREHMA-instrumented client was able to classify the replayed message as a valid signed message, since the `max-age=3600` directive declares that the cached signature is still in its defined validity time frame. These results show that CREHMA is compatible with common web caching systems when explicit freshness lifetime caching is deployed.

We performed a similar test to check the processing of CREHMA-protected messages with an explicit freshness validation. In this setup the client initiates the test procedure with a signed GET request. The server then sends a signed response including the `ETag` and `Cache-Control` headers, with the latter containing a value instructing the cache to validate the freshness (i.e. `no-cache`, `must-revalidate`, or `proxy-revalidate`). The client then issued subsequent signed GET requests and inspects the obtained response.

	Chrome	Safari	Firefox	Edge	ATS	Apache HTTPD	nginx	Squid	CloudFront	Cloudflare
Freshness Lifetime	●	●	●	●	●	●	●	●	●	●
Freshness Validation	●	●	●	●	●	●	○	●	○	○

Table 7.3: Compatibility evaluation of CREHMA with real world caches storing and reusing signed response messages

The results are shown in Table 7.3. They highlight what caches are compatible with the storage and reuse of CREHMA-signed response messages based on freshness validation caching. The client was able to successfully verify the cached response messages of Chrome, Firefox, Safari, Edge, ATS, Apache HTTPD, and Squid. These caches comply with the Web caching standard, as they update the header of the cached response when the freshness validation was successful. This is, however, not the case for nginx, CloudFront and Cloudflare. These web caching systems do not update the header of the cached response when a freshness validation is successful. The caches always replay the initially cached response. These observations are inline with the results presented in [NLF19a]. Here the authors showed that these particular caching systems do not behave compliant with the Web caching standard. Such a violation leads to the issue that reused signed messages are classified as a replay attack, as the signature freshness is expired. Service providers which intend to use CREHMA in conjunction with these caches, should therefore avoid using caching with freshness validation. Alternatively, they can exclusively use caching with explicit freshness lifetime or simply exclude such response messages from caching so that for each request message a new response message with a new `Signature` header is returned from the origin server.

## 7.7.2 Performance

The goal of this evaluation was to gain insights on the influence of CREHMA to the performance of a web application using caching. We therefore designed experiments to capture the message processing times of CREHMA-protected messages and compare those with equivalent setups containing no message protection mean and REHMA, the non-cache-aware equivalent to CREHMA. By message processing time we mean the time elapsed from the request being generated and issued by the client to the time the response has been received back by the client. This includes the generation of request and response messages, the parsing of request and response messages, and the round trip time. The test cases including HTTP signature schemes add to this with signature generation and verification times for both request and response messages.

All virtual machines were running in the EC2 instance type `t2.micro`. All our performance experiments followed this general setup. In one test run, the client requests a certain resource 200 times from the server with a request frequency of 1Hz. The cache is blanked for each test run. Depending on the test case, the origin server either sets an explicit freshness lifetime or an explicit freshness validation policy. In a test run, the test client captures the message processing times of all 200 requests. We conducted test runs with different response body sizes ranging from 1KB to 10MB. We performed our experiments with three different types of caches: a web browser cache (Chrome), a proxy cache (ATS) and a CDN (CloudFront). We deployed our test client, test server, and the real world cache in distinct regions of the AWS cloud.

## Freshness Lifetime Experiments Results

Figure 7.7 shows the results of the experiments with the explicit freshness lifetime policy set to `Cache-Control:max-age=3600`. Each plotted value represents the average of 200 measurements. According to our measurement results, CREHMA introduces only a slight delay compared to the unprotected case. We could also observe, that this delay increases with a growing body size. For response messages with a 1KB body the introduced delay is less than 1ms for all cache types. The processing time of signed response messages with a 10MB body took 44ms in average longer than in the unprotected case in Chrome. When using ATS, the difference was 42ms in average. In CloudFront the message processing time differs by 29ms in average. Our detailed measurements shows that the signature generation time of GET requests at the client-side and the signature verification of such requests at the server-side take less than 1ms in average each. This is due to the empty body in these messages. The signature generation time for response messages with a 10MB body took 40ms in average. The response message signature generation needs to be performed only once for the initial request, though, as the signed response message will be cached and served from the cache without further server intervention. Thus, the measured delay introduced by CREHMA is mainly caused by the hash generation of the body during response message signature verification. The measured differences in the distinct test case are due to distinct client implementations. For 10MB, the average hash calculation time in our Ruby client takes 29ms and 43ms for the Javascript client. Also, the message expansion introduced by CREHMA is minimal and boils down to a static offset defined by the size of the `Signature` header. In terms of our experiments this means 43B for the signature value, 24B for the TVP and some more bytes for additionally required metadata, such as algorithm names and a key ID. Although this static overhead is added to every CREHMA-protected request and response message, still the overall amount is rather small and does e.g. not lead to a measurable increase in data transfer time.

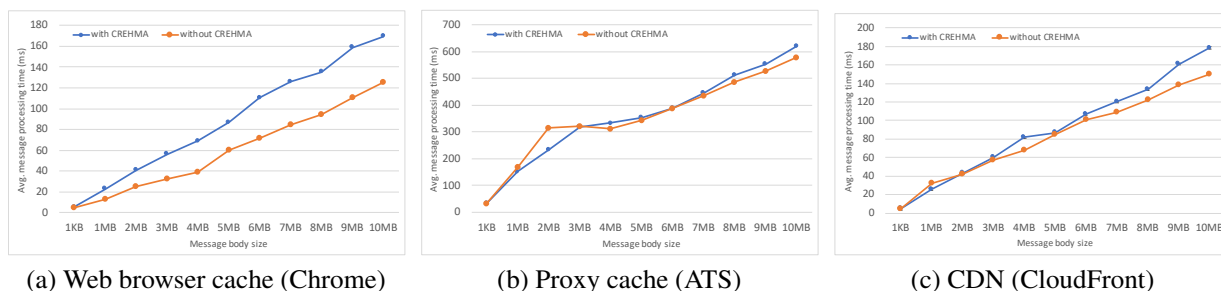


Figure 7.7: Comparison between the average processing time of cached messages with CREHMA protection and cached messages without CREHMA protection via explicit freshness lifetime.

Some measurements for small payload sizes showed that the average message processing time of unsigned messages is higher than for signed messages. As the measurements include the round trip times and we deployed our test beds in the AWS cloud, these deviations are most likely attributable to network fluctuations.

## Freshness Validation Experiments Results

Further experiments targeted caching policies adopting freshness validation. We conducted the test runs with the same general setup, only the `Cache-Control` header was assigned with the `no-cache` directive in the tests with Chrome and CloudFront. Since ATS does not support the `no-cache` directive properly [NLF19a], we had to use `max-age=0, must-revalidate` instead to force ATS to validate the freshness of each request message.

Figure 7.8 shows the results of our experiments in respect to caching with freshness validation. The gathered results are very similar to the once from the previous section. The average delay for retrieving a signed 1KB response message is 5ms with the Chrome browser cache, 2ms with the ATS proxy cache and less than 1ms with the CloudFront CDN. Likewise, we measured larger differences in message processing times when the response payload size increased. In Chrome, CREHMA induced a 51ms delay in average for response body size of 10MB. The average delay for the same body size when using the proxy cache ATS was 16ms. If CloudFront was used, the message processing of CREHMA-protected messages time took 57ms in average longer. In summary, we observed the processing time of CREHMA-protected messages is governed by the message body size, the programming environment and by the regional locations of the cache and endpoints. The message overhead added by the `Signature` and `Validation-Signature` header barely affects the performance of the communication.

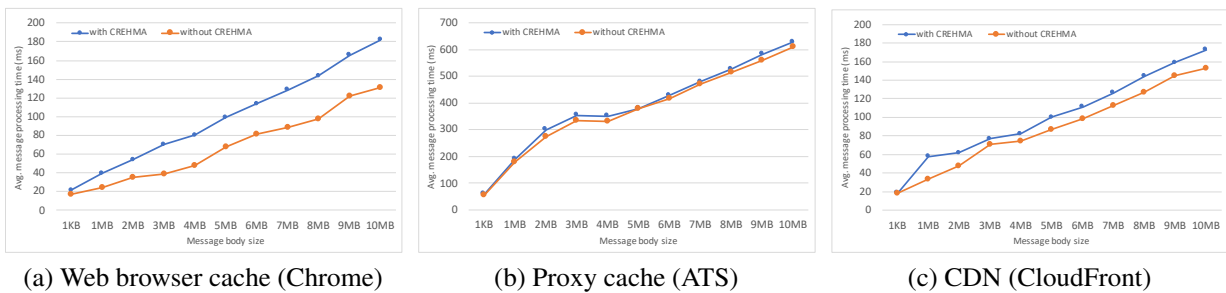


Figure 7.8: Comparison between the average processing time of cached messages with CREHMA protection and cached messages without CREHMA protection via freshness validation.

## Non-Cache-Aware HTTP Signature Schemes

Finally, we compare the performance of CREHMA and other comprehensive HTTP signature schemes that do not support caching in order to quantify possible advantages. From the available state of the art we chose REHMA. The schemes of Serme et al. and Cavage et al. can be considered as similar, as all three HTTP signature schemes differ only in terms of what headers they protected. The signature generation and verification procedures are similar. Also, caching has to be disabled for all three schemes, as the signed responses are not intended to be replayed by any entity including caches. We implemented a REHMA-instrumented prototype server and two REHMA-instrumented clients for testing proxy caches and CDNs following the implementation of the CREHMA testbeds. The REHMA-instrumented systems were deployed on the same EC2 instances as the CREHMA-instrumented systems.

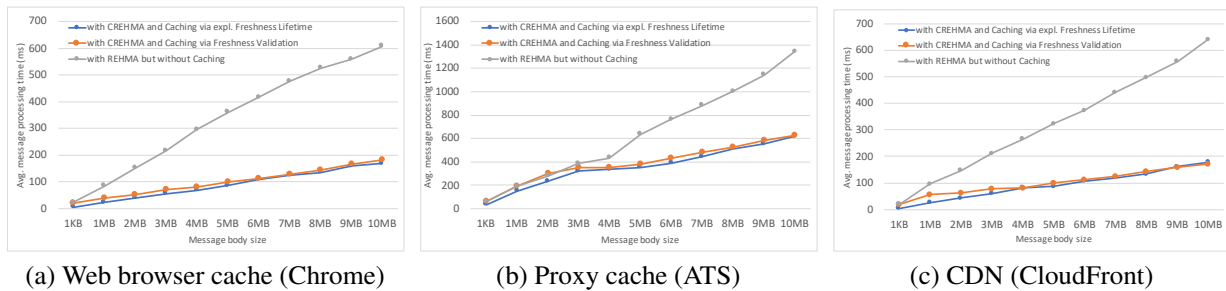


Figure 7.9: Comparison between the average processing time of cached messages with CREHMA protection and uncached messages with REHMA protection.

The results obtained from our experiments are shown in Figure 7.9. Since REHMA does not support caching, each request message is forwarded by the cache to the origin server, which returns a full response message including a body. When using CREHMA, cacheable response messages do not require to be retrieved from the origin server. They can be delivered directly by the cache reducing the amount of signature operations, end-to-end delay and data traffic. In case CREHMA is used with a browser cache such as contained in Chrome, the message processing time is up to three times faster for response messages with a 10MB body than with REHMA. This is also true for the CloudFront CDN. In the test run based on the ATS proxy cache, the message processing time for response messages with a 10MB body with CREHMA is twice as fast as than with REHMA.

Finally, our experiments showed that CREHMA can also be deployed and used in web-based systems that make no use of caching. This implies similar performance penalties as when using REHMA.

### 7.7.3 Security

The goal of CREHMA is to ensure end-to-end integrity and authenticity of HTTP messages while enabling cacheability. By this, CREHMA aims at mitigating the attack vectors identified by the evolved threat model introduced in Section 7.3 and at fulfilling the emerging security-related requirements for cache-aware HTTP signature schemes in Section 7.5. Here, we evaluate to what extent CREHMA achieves these security goals. Moreover, we broaden the scope of the security analysis by discussing known attack classes to Web caches, such as HTTP response splitting [Kle04] and HTTP request smuggling [Lin+05], in the light of CREHMA-enabled systems.

In general, due to the signature of the mandatory HTTP headers and the body of both the request and the response, a verifying party is able to detect any modification of the protected elements of these messages. This thwarts attack vectors of type (A1) in Section 7.3. As the CREHMA signature generation and verification algorithms check and enforce a strict message structure, related attacks such as signature wrapping [GL09] are defeated. Application-specific needs can be added to either the unprotected or the protected message areas according to the respective critically.

To mitigate replay attacks as depicted by (A2) and (A3) in Section 7.3, CREHMA adds a TVP to each signature generation which induces—amongst others—that each generated signature value is unique. Thus, a replay attack can be detected when a signature value is received twice.

To distinguish between a replay attack and a legitimate message reuse provided by benign cache, CREHMA requires to verify the signature freshness which can be derived from the response freshness as described in Section 7.6. Hence, CREHMA enables cacheability while effectively detecting replay attacks.

Moreover, with the integration of the cache key in the signature verification process, CREHMA allows to detect attack vectors in which a malicious cache swaps a signed response message with another signed response message (see (A4) in Section 7.3). Beside the detection of response message swapping attacks, the consideration of the cache key also mitigates a set of cache poisoning attacks *Response Splitting* [Kle04], *Request Smuggling* [Lin+05] and *Host of Trouble* [Che+16]. In these attack vectors, a malicious client aims to inject a malicious response message under the cache key of the target resource. This harmful content is then returned when a client requests the injected URL. When using CREHMA, the described web cache poisoning attacks can be detected, as an attacker-crafted response message does not contain a valid signature. There are also web cache poisoning attacks which intend to replace a response message with another response message of another resource endpoint from the same origin server such as *HTTP Desync attacks* [Ket19a]. If CREHMA is used such web cache poisoning attacks can be detected as well, since the signature value of the replaced response message includes another cache key as the genuine one.

## 7.8 Conclusion and Outlook

In this paper, we introduce CREHMA, a cache-aware HTTP signature scheme that provides comprehensive end-to-end integrity and authenticity for Web-based systems. The need for security mechanisms that complement TLS is introduced by an evolved threat model that considers intermediate systems in Web-based software systems. Our analyses show that CREHMA only causes minor delays and data extensions in the provision of its end-to-end security services. Moreover, CREHMA outperforms the existing HTTP signature systems both in terms of security as they provide no comprehensive protection and in terms of performance as they are not cacheable. Service providers can immediately use CREHMA with available Web caching systems, since CREHMA does not require any changes to caches.

In future work we will study the key management of CREHMA deployments as well as the confidentiality of CREHMA-protected messages. Moreover, we will analyze the performance and compatibility when CREHMA is integrated in a cache. This will enable to provide further benefits, such as the access control to cached private resources. Finally, we would like to understand to what extent the CREHMA approach can be generalized to meet the end-to-end security in Web-based distributed software systems to other intermediate systems including load balancers, application firewalls, HTTPS interceptors, protocol converters, and message routers.



# Chapter 8

## Summary and Further Work

Reliable software systems that scale at large are the driving forces of the digital transformation. The industry and academia have conducted many initiatives to explore the security and scalability of such systems [Fei+06]. This thesis comprises several works to study and enhance the security of REST-based ULS systems. It provides two comprehensive state-of-the-art analyses on current work in REST-based authentication schemes and service description languages. Based on this background, this work proposes a methodology on how to develop REST-Security components for any kind REST-based technologies. With this approach, REMA is introduced, a generic security scheme for ensuring end-to-end integrity and authenticity of REST messages. REMA then serves as a guideline to derive REHMA and RECMA, which provides end-to-end security for HTTP and CoAP messages. As intermediate systems are vital components for the scalability and security of modern distributed systems, a large-scale study on web caching has been conducted. The analysis revealed many malfunctions and non-conformances, which may lead to potential vulnerabilities. The developed cache testing tool and the whole test suite, which have been used for the analysis, can be downloaded as open-source and free software via GitHub (see Appendix A). In further investigations on web caching, CPDoS has been discovered, a novel web cache poisoning attack class. The experiments show that millions of websites are affected by the discovered attack. To mitigate the vulnerabilities, this thesis proposes and discusses countermeasures in cooperation with affected organizations. The term CPDoS has been established as a new class of attacks that exploit a cache to provoke a denial of service. Other researcher took the findings of this thesis as a role model to discover and report other CPDoS variations [Ket19b; Dav20; Dav19]. Moreover, many security and computer science news platforms reported the findings to inform organizations and people on CPDoS. More details on the media coverage and recent information on CPDoS can be found on the website <https://cpdos.org>. Based on the knowledge from the studies in web caching, this thesis proposes CREHMA, an extension of REHMA. Unlike available HTTP signature schemes, CREHMA ensures comprehensive end-to-end authenticity and integrity of HTTP messages without loss of cacheability and vice versa. As with the cache testing tool, software developers can download CREHMA as a free and open-source tool via GitHub. More details, can be found in at the Appendix B.

CREHMA is the first security scheme for REST, which takes caches into account to enable scalability. Security and scalability are two key quality factors in modern distributed systems. Caches ensure scalability by storing and recycling frequently used resources. In terms of security, caches provide increased availability. CDNs, which include a WAF, can protect against DDoS attacks and additionally filter malicious requests. These observations show that intermediate systems are vital elements for security and scalability. However, intermediate systems can also be



misused to impair the availability, as shown, e.g., by the introduced CPDoS attack. Researchers also demonstrated with, e.g., HTTP Request Smuggling [Lin+05] or Host of Trouble [Che+16], that they can distribute malicious content to millions of users using a cache [Ket19a; Che+16]. Moreover, the usage of intermediate systems interrupts the transport security provided by TLS. This all shows that even though intermediaries are crucial for any modern distributed system, the usage of them in real-world systems still entails severe vulnerabilities. The mitigation of security threats in layered systems requires, therefore, an in-depth understanding of the interplay between intermediaries and endpoints. This thesis contributes to these efforts.

Chapter 7 shows that CREHMA not only mitigates man-in-the-middle attacks but also detects web cache poisoning attacks, including HTTP Request Smuggling, HTTP Response Splitting, and Host of Trouble. However, CREHMA cannot thwart the presented CPDoS attacks and some of the web cache poisoning techniques of James Kettle [Ket18c] as the poisoned response is returned from the target origin server itself. To address these attacks, countermeasures need to be designed which filter the malicious requests before they reach the origin server.

Chapter 4 and Chapter 5 mainly focus on caches. Caches are not the only intermediate systems in REST-based applications. The interference of other intermediaries needs to be studied as well. Further work should conduct similar studies on, e.g., load balancer or WAF to explore the impact of other intermediate systems.

The end-to-end message confidentiality for REST messages is another important topic that is only partially addressed by this work. Chapter 2 specifies the requirements for designing a REST message confidentiality. To avoid malicious intermediaries reading sensitive information, it requires to define a policy describing what class of intermediate systems has read access to what type of message elements. Such a policy can be defined for caches at first. Chapter 4 and Chapter 5 already point out the cache-related headers. Based on studies on other intermediate systems, similar policies need to be specified for other intermediate systems such as load balancers or WAFs.

Another crucial topic that is not covered by this thesis is a CREHMA counterpart for CoAP. As CoAP is based on HTTP and specifies similar cache-related control directive, a CREHMA adaptation to CoAP might be very straightforward. Such a cache-ware signature scheme should be developed in further work to ensure end-to-end authenticity and integrity under consideration of caching in IoT environments.

Future work should conduct further studies in other mission-critical application domains of REST. The upcoming mobile 5G network for mobile and wireless devices is a REST-based ULS system which requires high as well as specific security demands. In the publication [Rud+19], my co-authors and myself already discussed the security challenges for REST-based services in 5G. Further studies need to address these challenges as well as evaluate CREHMA and cache-related security issues in 5G software systems.

## Final declaration / Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare, on oath, that I have written this dissertation by my own and have not used other than the acknowledged resources and aids.

Köln, den 30. Mai 2020

---

Hoai Viet Nguyen

# Bibliography

- [Aas+19] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and B. Warren. *Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web*. In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019. URL: <https://doi.org/10.1145/3319535.3363192>.
- [ABM15] B. Adida, M. Birbeck, and S. McCarron. *RDFa Core 1.1 - Third Edition*. W3C Recommendation. W3C, 2015. URL: <http://www.w3.org/TR/2015/REC-rdfa-core-20150317>.
- [Ama19a] Amazon. *How CloudFront Processes and Caches HTTP 4xx and 5xx Status Codes from Your Origin*. 2019. URL: <https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/HTTPStatusCodes.html>.
- [Ama19b] Amazon. *Signing AWS API Requests*. 2019. URL: [https://docs.aws.amazon.com/general/latest/gr/signing\\_aws\\_api\\_requests.html](https://docs.aws.amazon.com/general/latest/gr/signing_aws_api_requests.html).
- [Apa19] Apache HTTP Server Project. *Security Tips*. 2019. URL: [https://httpd.apache.org/docs/trunk/misc/security\\_tips.html](https://httpd.apache.org/docs/trunk/misc/security_tips.html).
- [Api16] Apiary Inc. *Markdown Syntax for Object Notation*. Tech. rep. 2016. URL: <https://github.com/apiaryio/mson>.
- [AW10] R. Alarcon and E. Wilde. *Linking Data from RESTful Services*. In: *Third Workshop on Linked Data on the Web*. 2010.
- [Bar+08] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML Signature Syntax and Processing (Second Edition)*. Recommendation. W3C, 2008. URL: <http://www.w3.org/TR/xmlsig-core/>.
- [BAS12] J. Bellido, R. Alarcon, and C. Sepulveda. *Web Linking-based Protocols for Guiding RESTful M2M Interaction*. In: *11th International Conference on Current Trends in Web Engineering (ICWE)*. 2012. ISBN: 978-3-642-27996-6. DOI: [10.1007/978-3-642-27997-3\\_7](https://doi.org/10.1007/978-3-642-27997-3_7). URL: [https://doi.org/10.1007/978-3-642-27997-3\\_7](https://doi.org/10.1007/978-3-642-27997-3_7).
- [BCS12] C. Bormann, A.P. Castellani, and Z. Shelby. *CoAP: An Application Protocol for Billions of Tiny Internet Nodes*. In: *IEEE Internet Computing* 16.2 (2012), pp. 62–67. ISSN: 1089-7801. URL: <https://10.1109/MIC.2012.29>.
- [BEN09] O. Ben-Kiki, C. Evans, and I. dot Net. *YAML Ain't Markup Language Version 1.2*. Tech. rep. 2009. URL: <http://www.yaml.org/spec/1.2/spec.html>.
- [Ben15] Benki. *CacheViewer*. 2015. URL: <https://addons.mozilla.org/de/firefox/addon/cacheviewer/>.

- [BFM05] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. IETF, 2005. URL: <https://tools.ietf.org/html/rfc3986>.
- [BH13] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. IETF, 2013. URL: <http://www.ietf.org/rfc/rfc7049.txt>.
- [Blu16] API Blueprint. *API Blueprint Specification*. 2016. URL: <https://apiblueprint.org/documentation/specification.html> (visited on 11/05/2016).
- [BM14] D. Brickely and L. Miller. *FOAF Vocabulary Specification 0.99*. Tech. rep. 2014. URL: <http://xmlns.com/foaf/spec/>.
- [BO00a] G. Barish and K. Obraczke. *World Wide Web caching: trends and techniques*. In: *IEEE Communications Magazine* 38.5 (2000), pp. 178–184. URL: <https://doi.org/10.1109/35.841844>.
- [BO00b] G. Barish and K. Obraczke. *World Wide Web caching: trends and techniques*. In: *IEEE Communications Magazine* 38.5 (2000), pp. 178–184. ISSN: 0163-6804. DOI: [10.1109/35.841844](https://doi.org/10.1109/35.841844). URL: <https://doi.org/10.1109/35.841844>.
- [Boo18] G. Booch. *The History of Software Engineering*. In: *IEEE Software* 35.5 (2018), pp. 108–114.
- [Bor17] C. Bormann. *Constrained Object Signing and Encryption (COSE)*. RFC 8152. IETF, 2017. URL: <https://tools.ietf.org/html/rfc8152>.
- [BPM18] S. Bennetts, R. Pereira, and R. Mitchell. *OWASP Zed Attack Proxy Project*. 2018. URL: [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project).
- [BPT15] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7540>.
- [Bra+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Recommendation. W3C, 2008. URL: <http://www.w3.org/TR/2008/REC-xml-20081126>.
- [Bra16] T. Bray. *An HTTP Status Code to Report Legal Obstacles*. RFC 7725. IETF, 2016. URL: <https://tools.ietf.org/html/rfc7725>.
- [Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. IETF, 2017. URL: <https://tools.ietf.org/html/rfc8259>.
- [BRS19] A. Backman, J. Richer, and M. Sporny. *Signing HTTP Messages draft-richanna-http-message-signatures-00*. Internet-Draft. IETF, 2019. URL: <https://tools.ietf.org/html/draft-richanna-http-message-signatures-00>.
- [BVR13] W. Bellante, R. Vilaridi, and D. Rossi. *On Netflix catalog dynamics and caching performance*. In: *IEEE 18th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 2013.
- [BW16] K. Brown and B. Woolf. *Implementation Patterns for Microservices Architectures*. In: *23rd Conference on Pattern Languages of Programs (PLoP)*. 2016.
- [Cal+19] S. Calzavara, R. Focardi, M. Nemeč, A. Rabitti, and M. Squarcina. *Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem*. In: *39th IEEE Symposium on Security and Privacy (S&P)*. 2019.

- [Car+05] L. G. Cardenas, J. A. Gil, J. Domenech, J. Sahuquillo, and A. Pont. *Performance comparison of a Web cache simulation framework*. In: *19th International Conference on Advanced Information Networking and Applications (AINA)*. 2005. URL: <https://doi.org/10.1109/AINA.2005.275>.
- [Car+18] M. Carr, E. Lupu, J. Norton, L. Smith, J. Blackstock, H. Boyes, A. Hudson-Smith, I. Brass, H. Chizari, R. Cooper, P. Coulton, B. Craggs, N. Davies, D. De Roure, M. Elsdon, M. Huth, J. Lindley, C. Marple, B. Mittelstadt, R. Nicolescu, J. Nurse, R. Proctor, P. Radanliev, A. Rashid, D. Sgandurra, A. Skatova, M. Taddeo, L. Tanczer, R. Vieira-Steiner, J. D. M. Watson, S. Wachter, S. Wakenshaw, G. Carvalho, R. J. Thompson, and P. S. Westbury. *Internet of Things: realising the potential of a trusted smart world*. Ed. by P. Taylor and S. Allpress. Royal Academy of Engineering, 2018.
- [CB02] B. Carpenter and S. Brim. *Middleboxes: Taxonomy and Issues*. RFC 3234. IETF, 2002. URL: <https://tools.ietf.org/html/rfc3234>.
- [Cha19] A. Chatiron. *Define allowed methods used in 'X-HTTP-Method-Override'*. 2019. URL: <https://github.com/playframework/play1/issues/1300>.
- [Che+16] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson. *Host of Troubles: Multiple Host Ambiguities in HTTP Implementations*. In: *23th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.
- [Chi+07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C Recommendation. W3C, 2007. URL: <http://www.w3.org/TR/2007/REC-wsdl20-20070626>.
- [Chr+00] Erik Christensen, Franciso Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C Note. W3C, 2000. URL: <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [CJ10] G. Clemm and J. Whitehead J. Crawford J. Reschke. *Binding Extensions to Web Distributed Authoring and Versioning (WebDAV)*. RFC 5842. IETF, 2010. URL: <https://tools.ietf.org/html/rfc5842>.
- [CS19] M. Cavage and M. Sporny. *Signing HTTP Messages*. Internet-Draft. IETF, 2019.
- [Dav19] N. Davison. *Abusing HTTP hop-by-hop request headers*. 2019. URL: <https://nathandavison.com/blog/abusing-http-hop-by-hop-request-headers>.
- [Dav20] N. Davison. *Cache poisoning DoS in CloudFoundry gorouter (CVE-2020-5401)*. 2020. URL: <https://nathandavison.com/blog/cache-poisoning-dos-in-cloudfoundry-gorouter>.
- [de +13] B. de Azevedo Muniz, L. M. Chaves, H. A. Lira, J. R. V. Dantas, and P. P. M. Farias. *SERIN – AN APROACH TO SPECIFY SEMANTIC ABSTRACT INTERFACES IN THE CONTEXT OF RESTFUL WEB SERVICES*. In: *IADIS International Conference WWW/INTERNET*. 2013.
- [De +14] F. De Backere, B. Hanssens, R. Heynssens, R. Houthoof, A. Zuliani, S. Verstichel, B. Dhoedt, and F. De Turck. *Design of a security mechanism for RESTful Web Service communication through mobile clients*. In: *IEEE Network Operations and Management Symposium (NOMS)*. 2014, pp. 1–6. URL: <https://doi.org/10.1109/NOMS.2014.6838308>.

- [Dev+16] A. Devdatta, F. Braun, F. Marier, and J. Weinberger. *Subresource Integrity*. W3C Recommendation. W3C, 2016. URL: <https://www.w3.org/TR/SRI/>.
- [DS10] L. Dusseault and J. Snell. *PATCH Method for HTTP*. RFC 5789. IETF, 2010. URL: <https://tools.ietf.org/html/rfc5789>.
- [Dur+17] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. *The Security Impact of HTTPS Interception*. In: *24th Network and Distributed Systems Symposium (NDSS)*. 2017. URL: [https://www.internetsociety.org/sites/default/files/ndss2017%5C\\_04A-4%5C\\_Durumeric%5C\\_paper.pdf](https://www.internetsociety.org/sites/default/files/ndss2017%5C_04A-4%5C_Durumeric%5C_paper.pdf).
- [Dus07] L. Dusseault. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. RFC 4918. IETF, 2007.
- [EPM13a] T. Erl, R. Puttini, and Z. Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education, 2013. ISBN: 9780133387513.
- [EPM13b] T. Erl, R. Puttini, and Z. Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education, 2013. ISBN: 9780133387513.
- [Erl+13] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. The Prentice Hall service technology series. Prentice Hall, 2013, pp. I–XXXII, 1–577. ISBN: 978-0-13-701251-0.
- [Erl07] T. Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007. ISBN: 0132344823.
- [Fah+12] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. *Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security*. In: *19th ACM Conference on Computer and Communications Security (CCS)*. 2012. DOI: [10.1145/2382196.2382205](https://doi.org/10.1145/2382196.2382205). URL: <http://doi.acm.org/10.1145/2382196.2382205>.
- [Fei+06] P. Feiler, K. Sullivan, K. Wallnau, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, and D. Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.
- [Fel+17] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. *Measuring HTTPS adoption on the web*. In: *26th USENIX Security Symposium*. 2017.
- [FHT15] S. Farrell, P. Hoffman, and M. Thomas. *HTTP Origin-Bound Authentication (HOBA)*. Experimental RFC 7486. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7486>.
- [Fie+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. IETF, 1999.
- [Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis. University of California, Irvine, 2000. URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [Fla10] Flask. *Adding HTTP Method Overrides*. 2010. URL: <http://flask.pocoo.org/docs/1.0/patterns/methodoverrides/>.
- [FNR14] R. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7234>.



- [FR14a] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7232>.
- [FR14b] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7230>.
- [FR14c] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7231>.
- [Geo+12] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. *The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software*. In: *19th ACM Conference on Computer and Communications Security (CCS)*. 2012. ISBN: 978-1-4503-1651-4. DOI: [10.1145/2382196.2382204](https://doi.org/10.1145/2382196.2382204). URL: <https://doi.org/10.1145/2382196.2382204>.
- [Gil+16] Y. Gilad, A. Herzberg, M. Sudkovitch, and M. Goberman. *CDN-on-Demand: An affordable DDoS Defense via Untrusted Clouds*. In: *Network and Distributed System Security Symposium (NDSS)*. 2016. DOI: [10.14722/ndss.2016.23109](https://doi.org/10.14722/ndss.2016.23109). URL: <https://doi.org/10.14722/ndss.2016.23109>.
- [Gil17] O. Gil. *WEB CACHE DECEPTION ATTACK*. In: *Blackhat USA*. Black Hat USA, 2017. URL: <https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html>.
- [GL09] N. Gruschka and L. Lo Iacono. *Vulnerable Cloud: SOAP Message Security Validation Revisited*. In: *IEEE International Conference on Web Services*. 2009.
- [GL16] P. L. Gorski and L. Lo Iacono. *Towards the Usability Evaluation of Security APIs*. In: *10th International Symposium on Human Aspects of Information Security and Assurance (HAISA)*. 2016. URL: <https://cscan.org/openaccess/?id=287>.
- [GMS13] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. *Application-Layer Security for the WoT: Extending CoAP to Support End-to-End Message Security for Internet-Integrated Sensing Applications*. In: *11th International Conference on Wired & Wireless Internet Communications*. 2013. URL: [https://doi.org/10.1007/978-3-642-38401-1%5C\\_11](https://doi.org/10.1007/978-3-642-38401-1%5C_11).
- [GN09] Marc Goodner and Anthony Nadalin. *Web Services Federation Language (WS-Federation) Version 1.2*. Standard. Version 1.2. OASIS, 2009. URL: <http://docs.oasis-open.org/wsfed/federation/v1.2/ws-federation.html>.
- [Goo17] Google. *Migrating from Amazon S3 to Google Cloud Storage*. 2017. URL: <https://cloud.google.com/storage/docs/migrating>.
- [Goo18] Google. *Puppeteer*. 2018. URL: <https://github.com/GoogleChrome/puppeteer>.
- [Gor+14a] P. L. Gorski, L. Lo Iacono, H. V. Nguyen, and D. B. Torkian. *Service Security Revisited*. In: *11th IEEE International Conference on Services Computing (SCC)*. 2014, pp. 464–471. URL: <https://doi.org/10.1109/SCC.2014.68>.
- [Gor+14b] P. L. Gorski, L. Lo Iacono, H. V. Nguyen, and D. B. Torkian. *SOA-Readiness of REST*. In: *3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer International Publishing, 2014. URL: [https://doi.org/10.1007/978-3-662-44879-3\\_6](https://doi.org/10.1007/978-3-662-44879-3_6).



- [GR15] H. Gimpel and M. Röglinger. *DIGITAL TRANSFORMATION: CHANGES AND CHANCES – Insights based on an Empirical Study*. 2015. URL: [https://www.fim-rc.de/wp-content/uploads/Fraunhofer-Studie\\_Digitale-Transformation.pdf](https://www.fim-rc.de/wp-content/uploads/Fraunhofer-Studie_Digitale-Transformation.pdf).
- [Gra+11] S. Graf, V. Zholudev, L. Lewandowski, and M. Waldvogel. *Hecate, Managing Authorization with RESTful XML*. In: *2nd International Workshop on RESTful Design (WS-REST)*. 2011. DOI: [10.1145/1967428.1967442](https://doi.org/10.1145/1967428.1967442). URL: <http://doi.acm.org/10.1145/1967428.1967442>.
- [GS16] M. Green and M. Smith. *Developers are Not the Enemy!: The Need for Usable Security APIs*. In: *IEEE Security Privacy 14.5* (2016), pp. 40–46. URL: <http://doi.org/10.1109/MSP.2016.111>.
- [Gud+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. W3C, 2007. URL: <http://www.w3.org/TR/soap12-part1/>.
- [Guo+18] R. Guo, J. Chen, B. Liu, J. Zhang, C. Zhang, H. Duan, T. Wan, J. Jiang, S. Hao, and Y. Jia. *Abusing CDNs for Fun and Profit: Security Issues in CDNs’ Origin Validation*. In: *IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 2018. URL: <https://doi.org/10.1109/SRDS.2018.00011>.
- [Ham10] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849. IETF, 2010. URL: <https://tools.ietf.org/html/rfc5849>.
- [Han+16a] R. Handl, R. Jeyaraman, M. Pizzo, and M. Biamonte. *OData JSON Format Version 4.0 Plus Errata 03*. OASIS Standard. OASIS, 2016. URL: <https://docs.oasis-open.org/odata/odata-json-format/v4.0/odata-json-format-v4.0.html>.
- [Han+16b] R. Handl, R. Jeyaraman, M. Pizzo, and M. Zurmuehl. *OData Version 4.0. Part 1: Protocol Plus Errata 03*. OASIS Standard. OASIS, 2016. URL: <https://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.html>.
- [Har+13] B. Hartel, R. Jeyaraman, M. Zurmuehl, M. Pizzo, and R. Handl. *OData Atom Format Version 4.0*. OASIS Standard. OASIS, 2013. URL: <https://docs.oasis-open.org/odata/odata-atom-format/v4.0/odata-atom-format-v4.0.html>.
- [Har12] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6749>.
- [Hea09] Marc Headley. *Web Application Description Language (WADL)*. W3C Member Submission. W3C, 2009. URL: <http://www.w3.org/Submission/2009/SUBM-wadl-20090831>.
- [Hed+18] R. Hedberg, S. Gulliksson, M. Jones, and J. Bradley. *OpenID Connect Federation 1.0 - draft 04*. Draft. OpenID, 2018. URL: [https://openid.net/specs/openid-connect-federation-1\\_0.html](https://openid.net/specs/openid-connect-federation-1_0.html).
- [Her+] O. Hering, S. Calhoon, L. LaSeur, B. Poulos, O. Katz, and R. Towne. *2020 State of the Internet / Security: Financial Services – Hostile Takeover Attempts*. Tech. rep. Akamai. URL: <https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-financial-services-hostile-takeover-attempts-report-2020.pdf>.
- [Hew14] Hewlett Packard. *HP Helion Public Cloud Object Storage API Specification*. 2014. URL: <https://docs.hpcloud.com/publiccloud/api/object-storage/>.

- [Hic+14] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O’Connor, and S. Pfeiffer. *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. Recommendation. W3C, 2014. URL: <http://www.w3.org/TR/html5/>.
- [Hic16] I. Hickson. *Web Storage (Second Edition)*. W3C Recommendation. 2016. URL: <https://www.w3.org/TR/webstorage/>.
- [HM98] K. Holtman and A. Mutz. *Transparent Content Negotiation in HTTP*. RFC 2295. IETF, 1998. URL: <https://tools.ietf.org/html/rfc2295>.
- [HSS15] Ralph Holz, Yaron Sheffer, and Peter Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. RFC 7457. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7457>.
- [IEE18] IEEE Spectrum. *Interactive: The Top Programming Languages 2018*. 2018. URL: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018>.
- [IET14] IETF HTTP Working Group. *HTTP/1.1 Specification Compliance Testing*. 2014. URL: <https://github.com/http2/http2-test/>.
- [IET17] IETF JOSE Working Group. *Javascript Object Signing and Encryption (JOSE)*. 2017. URL: <http://datatracker.ietf.org/wg/jose/>.
- [Ima+13] T. Imamura, B. Dillaway, E. Simon, Y. Kelvin, and M. Nyström. *XML Encryption Syntax and Processing Version 1.1*. Recommendation. W3C, 2013. URL: <http://www.w3.org/TR/xmlenc-core1/>.
- [JBS15] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7515>.
- [JH15] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7516>.
- [Jia+15] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang. *Man-in-the-browser-cache*. In: *Computers and Security* 55.C (2015), pp. 62–80. ISSN: 0167-4048. DOI: [10.1016/j.cose.2015.07.004](https://doi.org/10.1016/j.cose.2015.07.004). URL: <http://dx.doi.org/10.1016/j.cose.2015.07.004>.
- [Jos06] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. IETF, 2006. URL: <https://tools.ietf.org/html/rfc4648>.
- [JS12] Suman Jana and Vitaly Shmatikov. *Abusing File Processing in Malware Detectors for Fun and Profit*. In: *33rd IEEE Symposium on Security and Privacy*. 2012, pp. 80–94. ISBN: 978-0-7695-4681-0. DOI: [10.1109/SP.2012.15](https://doi.org/10.1109/SP.2012.15). URL: <https://doi.org/10.1109/SP.2012.15>.
- [KC08] R. Kanneganti and P. Chodavarapu. *Soa Security*. Greenwich, CT, USA: Manning Publications Co., 2008. ISBN: 9781932394689.
- [Ket18a] J. Kettle. *Bypassing Web Cache Poisoning Countermeasures*. 2018. URL: <https://portswigger.net/blog/practical-web-cache-poisoning>.
- [Ket18b] J. Kettle. *Denial of service via cache poisoning*. 2018. URL: <https://hackerone.com/reports/409370>.
- [Ket18c] J. Kettle. *Practical Web Cache Poisoning*. In: *Black Hat USA*. 2018. URL: <https://portswigger.net/blog/practical-web-cache-poisoning>.
- [Ket19a] J. Kettle. *HTTP Desync Attacks: Smashing into the Cell Next Door*. In: *Black Hat USA*. 2019.

- [Ket19b] J. Kettle. *Responsible denial of service with web cache poisoning*. 2019. URL: <https://portswigger.net/research/responsible-denial-of-service-with-web-cache-poisoning>.
- [KGV08] J. Kopecký, K. Gomadam, and T. Vitvar. *hRESTS: An HTML Microformat for Describing RESTful Web Services*. In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 2008. DOI: [10.1109/WIIAT.2008.379](https://doi.org/10.1109/WIIAT.2008.379). URL: <https://doi.org/10.1109/WIIAT.2008.379>.
- [KJ10] M. Krizevnik and M. B. Juric. *Improved SOA Persistence Architectural Model*. In: *SIGSOFT Softw. Eng. Notes* 35.3 (May 2010). ISSN: 0163-5948.
- [Kle04] A. Klein. *Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics*. White Paper. Sanctum, Inc., 2004. URL: [https://dl.packetstormsecurity.net/papers/general/whitepaper\\_httpresponse.pdf](https://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf).
- [KR97] Rohit Khare and Adam Rifkin. *Weaving a Web of Trust*. In: *World Wide Web J.* 2.3 (June 1997), pp. 77–112. ISSN: 1085-2301. URL: <http://dl.acm.org/citation.cfm?id=275079.275089>.
- [Kru+17] S. R. Krueger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath. *CogniCrypt: Supporting developers in using cryptography*. In: *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017).
- [Lan13] M. Lanthaler. *Creating 3rd Generation Web APIs with Hydra*. In: *22nd International Conference on World Wide Web (WWW)*. 2013. DOI: [10.1145/2487788.2487799](https://doi.org/10.1145/2487788.2487799). URL: <https://doi.org/10.1145/2487788.2487799>.
- [LC11] L. Li and W. Chou. *Design and Describe REST API without Violating REST: A Petri Net Based Approach*. In: *18th IEEE International Conference on Web Services (ICWS)*. 2011. DOI: [10.1109/ICWS.2011.54](https://doi.org/10.1109/ICWS.2011.54).
- [LCB16] A. Levy, H. Corrigan-Gibbs, and D. Boneh. *Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser*. In: *IEEE Security Privacy* 14.2 (2016), pp. 22–28.
- [Leo16] S. Leonard. *Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations*. RFC. IETF, 2016. URL: <https://tools.ietf.org/html/rfc7764>.
- [Lew+07] Amelia Lewis, Hugo Haas, David Orchard, Sanjiva Weerawarana, Roberto Chinnici, and Jean-Jacques Moreau. *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*. W3C Recommendation. W3C, 2007.
- [LG17] L. Lo Iacono and P. L. Gorski. *I Do and I Understand. Not Yet True for Security APIs. So Sad*. In: *2nd European Workshop on Usable Security (EuroUSEC)*. 2017. URL: <https://doi.org/10.14722/eurosec.2017.23015>.
- [Lin+05] C. Linhart, A. Klein, R. Heled, and S. Orrin. *HTTP REQUEST SMUGGLING*. Whitepaper. 2005. URL: <http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf>.
- [LJK15] S. Lee, J.-Y. Jo, and Y. Kim. *A Method for Secure RESTful Web Service*. In: *IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*. 2015.

- [LJK17] S. Lee, J.-Y. Jo, and Y. Kim. *Authentication system for stateless RESTful Web service*. In: 17.S1 (2017), pp. 21–34.
- [LN15a] L. Lo Iacono and H. V. Nguyen. *Authentication Scheme for REST*. In: *International Conference on Future Network Systems and Security (FNSS)*. Springer International Publishing, 2015. URL: [https://doi.org/10.1007/978-3-319-19210-9\\_8](https://doi.org/10.1007/978-3-319-19210-9_8).
- [LN15b] L. Lo Iacono and H. V. Nguyen. *Towards Conformance Testing of REST-based Web Services*. In: *11th International Conference on Web Information Systems and Technologies (WEBIST)*. 2015. URL: <https://doi.org/10.5220/0005412202170227>.
- [LNG19] L. Lo Iacono, H. V. Nguyen, and P. L. Gorski. *On the Need for a General REST-Security Framework*. In: *Future Internet* 11.3 (2019). URL: <https://doi.org/10.3390/fi11030056>.
- [Lo +14] L. Lo Iacono, H. V. Nguyen, T. Hirsch, M. Baiers, and S. Möller. *UI-Dressing to Detect Phishing*. In: *IEEE 6th International Symposium on Cyberspace Safety and Security (CSS)*. 2014. URL: <https://dx.doi.org/10.1109/HPCC.2014.126>.
- [Lov17] Jacob Loveless. *Cache Me If You Can*. In: *Queue* 15.4 (2017). ISSN: 1542-7730.
- [LRS02] F. Leymann, D. Roller, and M.-T. Schmidt. *Web Services and Business Process Management*. In: *IBM Systems Journal* 41.2 (2002), pp. 198–211. ISSN: 0018-8670. DOI: [10.1147/sj.412.0198](https://doi.org/10.1147/sj.412.0198). URL: <https://doi.org/10.1147/sj.412.0198>.
- [LS14] K. Li and R. Sun. *CoAP Payload-Length Option Extension*. Internet-Draft. IETF, 2014. URL: <https://tools.ietf.org/html/draft-li-core-coap-payload-length-option-03>.
- [ML17] M. Lanthaler. *Hydra Core Vocabulary - A Vocabulary for Hypermedia-Driven Web APIs*. Unofficial Draft. W3C, 2017. URL: <http://www.hydra-cg.com/spec/latest/core/>.
- [MS14] M. Sporny and D. Longley and G. Kellogg and M. Lanthaler and N. Lindström. *JSON-LD 1.0 - A JSON-based Serialization for Linked Data*. W3C Recommendation. W3C, 2014. URL: <https://www.w3.org/TR/json-ld/>.
- [Mal+10] M. Maleshkova, C. Pedrinaci, J. Domingue, G. Alvaro, and I. Martinez. *Using Semantics for Automating the Authentication of Web APIs*. In: *9th International Semantic Web Conference (ISWC)*. 2010. DOI: [10.1007/978-3-642-17746-0\\_34](https://doi.org/10.1007/978-3-642-17746-0_34). URL: [https://doi.org/10.1007/978-3-642-17746-0\\_34](https://doi.org/10.1007/978-3-642-17746-0_34).
- [Mao+] Y. Mao, L. Yong, L. Bo, J. Depeng, and C. Sheng. *Service-oriented 5G network architecture: an end-to-end software defining approach*. In: *International Journal of Communication Systems* 29.10 (), pp. 1645–1657. DOI: [10.1002/dac.2941](https://doi.org/10.1002/dac.2941). URL: <https://doi.org/10.1002/dac.2941>.
- [Mas98] L. Masinter. *Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)*. RFC 2324. IETF, 1998. URL: <https://tools.ietf.org/html/rfc2324>.
- [Mel16] A. Melnikov. *Salted Challenge Response HTTP Authentication Mechanism*. Experimental RFC 7804. IETF, 2016. URL: <https://tools.ietf.org/html/rfc7804>.
- [Mic17] Microsoft. *Authentication for the Azure Storage Services*. 2017. URL: <http://msdn.microsoft.com/en-us/library/dd179428.aspx>.
- [Nad+06] A. Nadalin, C. Kaler, R. Monzillo, and H.-B. Phillip. *Web Services Security: SOAP Message Security 1.1*. Standard. OASIS, 2006. URL: <http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>.

- [Nad+07] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. *WS-Trust 1.3*. Standard. OASIS, 2007. URL: <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html>.
- [Nad+09] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. *WS-SecureConversation 1.4*. Standard. OASIS, 2009. URL: <http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.html>.
- [Nad+12] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, and H. Granqvist. *WS-SecurityPolicy 1.3*. Standard. OASIS, 2012. URL: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/errata01/ws-securitypolicy-1.3-errata01-complete.html>.
- [NAT10] NATIONAL VULNERABILITY DATABASE. *CVE-2010-2730 Detail*. CVE 2010-2730. Nist, 2010. URL: <https://nvd.nist.gov/vuln/detail/CVE-2010-2730>.
- [NAT19] NATIONAL VULNERABILITY DATABASE. *CVE-2019-0941 Detail*. CVE 2019-0941. Nist, 2019. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-0941>.
- [Net19] Netcraft. *January 2019 Web Server Survey*. 2019. URL: <https://news.netcraft.com/archives/2019/01/24/january-2019-web-server-survey.html>.
- [New15] S. Newman. *Building microservices: designing fine-grained systems*. O'Reilly, 2015.
- [NF12] M. Nottingham and R. Fielding. *Additional HTTP Status Codes*. RFC 6585. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6585>.
- [Nir18] NirSoft. *ChromeCacheView*. 2018. URL: [https://www.nirsoft.net/utils/chrome\\_cache\\_view.html](https://www.nirsoft.net/utils/chrome_cache_view.html).
- [NL00] H. Nielsen and S. Lawrence. *An HTTP Extension Framework*. RFC 2774. IETF, 2000. URL: <https://tools.ietf.org/html/rfc2774>.
- [NL15] H. V. Nguyen and L. Lo Iacono. *REST-ful CoAP Message Authentication*. In: *International Workshop on Secure Internet of Things (SIoT), in conjunction with the European Symposium on Research in Computer Security (ESORICS)*. 2015. URL: <https://dx.doi.org/10.1109/SIoT.2015.8>.
- [NL16] H. V. Nguyen and L. Lo Iacono. *RESTful IoT Authentication Protocols*. In: *Mobile Security and Privacy - Advances, Challenges and Future Research Directions*. 1st ed. Advanced Topics in Information Security. Elsevier/Syngress, 2016, pp. 217–234. URL: <https://doi.org/10.1016/B978-0-12-804629-6.00010-9>.
- [NL20] H. V. Nguyen and L. Lo Iacono. *CREHMA: Cache-ware REST-ful Authentication Scheme*. In: *10th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2020. URL: <https://doi.org/10.1145/3374664.3375750>.
- [NLF18] H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Systematic Analysis of Web Browser Caches*. In: *2nd International conference on Web Studies (WS)*. 2018. URL: <https://doi.org/10.1145/3240431.3240443>.
- [NLF19a] H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Mind the Cache: Large-Scale Analysis of Web Caching*. In: *34rd ACM/SIGAPP Symposium on Applied Computing (SAC)*. 2019. URL: <https://doi.org/10.1145/3297280.3297526>.
- [NLF19b] H. V. Nguyen, L. Lo Iacono, and H. Federrath. *Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack*. In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019. URL: <https://doi.org/10.1145/3319535.3354215>.



- [Not19] M. Nottingham. *HTTP Caching Tests*. 2019. URL: <https://cache-tests.fyi/>.
- [NTL17] H. V. Nguyen, J. Tolsdorf, and L. Lo Iacono. *On the Security Expressiveness of REST-Based API Definition Languages*. In: *International Conference on Trust and Privacy in Digital Business (TrustBus)*. 2017. URL: [https://doi.org/10.1007/978-3-319-64483-7\\_14](https://doi.org/10.1007/978-3-319-64483-7_14).
- [Oiw+17a] Y. Oiwa, H. Takagi, K. Maeda, T. Hayashi, and Y. Ioku. *Mutual Authentication Protocol for HTTP*. Experimental RFC 8120. IETF, 2017. URL: <https://tools.ietf.org/html/rfc8120>.
- [Oiw+17b] Y. Oiwa, H. Takagi, K. Maeda, T. Hayashi, and Y. Ioku. *Mutual Authentication Protocol for HTTP: Cryptographic Algorithms Based on the Key Agreement Mechanism 3 (KAM3)*. Experimental RFC 8121. IETF, 2017. URL: <https://tools.ietf.org/html/rfc8121>.
- [Ope16] Open API Initiative. *OpenAPI Specification*. 2016. URL: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md>.
- [OWA17] OWASP. *Denial of Service Cheat Sheet*. 2017. URL: [https://www.owasp.org/index.php/Denial\\_of\\_Service\\_Cheat\\_Sheet%5C#Mitigation\\_3:\\_Limit\\_length\\_and\\_size](https://www.owasp.org/index.php/Denial_of_Service_Cheat_Sheet%5C#Mitigation_3:_Limit_length_and_size).
- [PCA16] V. Prokhorenko, K.-K. R. Choo, and H. Ashman. *Web application protection techniques: A taxonomy*. In: *Journal of Network and Computer Applications* 60 (2016), pp. 95–112. URL: <http://www.sciencedirect.com/science/article/pii/S1084804515002908>.
- [PLH09] D. Peng, C. Li, and H. Huo. *An extended UsernameToken-based approach for REST-style Web Service Security Authentication*. In: *2nd IEEE International Conference on Computer Science and Information Technology*. 2009. DOI: [10.1109/ICCSIT.2009.5234805](https://doi.org/10.1109/ICCSIT.2009.5234805).
- [Por18] PortSwigger. *Burp Suite Scanner*. 2018. URL: <https://portswigger.net/burp/>.
- [PTG14] Federica Paganelli, Stefano Turchi, and Dino Giuli. *A web of things framework for restful applications and its experimentation in a smart city*. In: *IEEE Systems Journal* 10.4 (2014), pp. 1412–1423.
- [Raj+10] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. *Cyber-physical systems: The next computing revolution*. In: *Design Automation Conference*. 2010.
- [RAM16] RAML. *RAML Version 1.0: RESTful API Modeling Language*. 2016. URL: <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md> (visited on 01/13/2017).
- [RBT16] J. Richer, J. Bradley, and H. Tschofenig. *A Method for Signing an HTTP Requests for OAuth*. Internet-Draft. IETF, 2016. URL: <https://tools.ietf.org/html/draft-ietf-oauth-signed-http-request-03>.
- [Ree15] A. Reemal. *HTTP/1.1 Specification Compliance Testing*. 2015. URL: <http://rimmythepaperclip.blogspot.com/2015/04/http11-specification-compliance-testing.html>.
- [Res15] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7617>.
- [Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF, 2018. URL: <https://tools.ietf.org/html/rfc8446>.

- [RLB03] Sean C. Rhea, Kevin Liang, and Eric Brewer. *Value-based Web Caching*. In: *12th International Conference on World Wide Web (WWW)*. ACM, 2003. URL: <http://doi.acm.org/10.1145/775152.775239>.
- [RM12] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6347>.
- [RMT14] J. Richer, W. Mills, and H. Tschofenig. *OAuth 2.0 Message Authentication Code (MAC) Tokens*. Internet-Draft. IETF, 2014. URL: <http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-05>.
- [Rob+13] Jonathan Robie, Rob Cavicchio, Rémon Sinnema, and Erik Wilde. *RESTful Service Description Language (RSDL): Describing RESTful Services Without Tight Coupling*. In: *Balisage: The Markup Conference 2013*. Montréal, Canada, 2013. DOI: DOI:10.4242/BalisageVol10.Robie01.
- [RR04] J. Rosenberg and D. Remy. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*. Pearson Higher Education, 2004. ISBN: 0672326515.
- [RR06] David Recordon and Drummond Reed. *OpenID 2.0: A Platform for User-centric Identity Management*. In: *2nd ACM Workshop on Digital Identity Management (DIM)*. 2006. DOI: 10.1145/1179529.1179532. URL: <https://doi.org/10.1145/1179529.1179532>.
- [RR08] L. Richardson and S. Ruby. *RESTful web services*. O’Reilly Media, Inc., 2008.
- [RSZ16] Jonathan Robie, Remon Sinnema, and William Zhou. *RESTful API Description Language*. 2016. URL: <https://github.com/restful-api-description-language>.
- [Rud+19] H. C. Rudolph, A. Kunz, L. Lo Iacono, and H. V. Nguyen. *Security Challenges of the 3GPP 5G Service Based Architecture*. In: *IEEE Communications Standards Magazine* 3.1 (2019), pp. 60–65. URL: <https://doi.org/10.1109/MCOMSTD.2019.1800034>.
- [Run19] A. Rundgren. *Signed HTTP Requests (SHREQ)*. Internet-Draft. IETF, 2019.
- [SAB15a] C. Sepulveda, R. Alarcon, and J. Bellido. *QoS aware descriptions for RESTful service composition: security domain*. In: *World Wide Web* 18.4 (2015), pp. 767–794. URL: <https://doi.org/10.1007/s11280-014-0278-0>.
- [SAB15b] R. Shekh-Yusef, D. Ahrens, and S. Bremer. *HTTP Digest Access Authentication*. RFC 7616. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7616>.
- [Sak+14] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0*. Specification. OpenID Foundation, 2014. URL: [http://openid.net/specs/openid-connect-core-1%5C\\_0.html](http://openid.net/specs/openid-connect-core-1%5C_0.html).
- [SB12] S.-T. Sun and K. Beznosov. *The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems*. In: *19th ACM Conference on Computer and Communications Security (CSS)*. 2012. URL: <http://doi.acm.org/10.1145/2382196.2382238>.
- [Sch17] Klaus Schwab. *The Fourth Industrial Revolution*. Crown Publishing Group, 2017. ISBN: 1524758868, 9781524758868.
- [Sch18] E. Schechter. 2018. URL: <https://www.blog.google/products/chrome/milestone-chrome-security-marking-http-not-secure/>.



- [Sel+18] G. Selander, J. Mattson, F. Palombini, and L. Seitz. *Object Security for Constrained RESTful Environments (OSCORE)*. Internet-Draft. IETF, 2018. URL: <https://tools.ietf.org/html/draft-ietf-core-object-security-15>.
- [Sel18] Selenium Contributors. *Selenium - Web Browser Automation*. 2018. URL: <https://www.seleniumhq.org/>.
- [Ser+12] G. Serme, A. S. De Oliveira, J. Massiera, and Y. Roudier. *Enabling message security for RESTful services*. In: *19th IEEE International Conference on Web Services (ICWS)*. 2012. URL: <https://doi.org/10.1109/ICWS.2012.94>.
- [SH13] H. Story and M. Hausenblas. *WebID specifications*. W3C Editor’s Draft. W3C, 2013. URL: <https://www.w3.org/2005/Incubator/webid/spec/>.
- [Sha05] T. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. IETF, 2005. URL: <https://tools.ietf.org/html/rfc4180>.
- [SHB] E. Stark, M. Hamburg, and D. Boneh. *Stanford Javascript Crypto Library*. URL: <http://bitwiseshiftleft.github.io/sjcl/>.
- [SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7252>.
- [Sma16] SmartBear Software. *Swagger Specification*. 2016. URL: <http://swagger.io/specification>.
- [Som+11] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono. *All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces*. In: *3rd ACM Workshop on Cloud Computing Security Workshop*. 2011.
- [SP18] C.-A. Staicu and M.I Pradel. *Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-based Web Servers*. In: *27th USENIX Conference on Security Symposium (USENIX Security)*. Baltimore, MD, USA: USENIX Association, 2018, pp. 361–376. ISBN: 978-1-931971-46-1. URL: <http://dl.acm.org/citation.cfm?id=3277203.3277231>.
- [Ste18] D. Stenberg. *curl - command line tool and library for transferring data with URLs*. 2018. URL: <https://curl.haxx.se/>.
- [Sto+09] H. Story, B. Harbulot, I. Jacobi, and M. Jones. *FOAF+SSL: RESTful Authentication for the Social Web*. In: *6th European Semantic Web Conference*. 2009.
- [T B11] T. Berners-Lee and D. Connolly. *Notation3 (N3): A readable RDF syntax*. W3C Team Submission. W3C, 2011.
- [TAR09] S. Triukosea, Z. Al-Qudad, and M. Rabinovich. *Content Delivery Networks: Protection or Threat?* In: *14th European Symposium on Research in Computer Security (ESORICS)*. 2009. URL: [https://doi.org/10.1007/978-3-642-04444-1\\_23](https://doi.org/10.1007/978-3-642-04444-1_23).
- [Tel18] Telerik. *Telerik Fiddler - The free web debugging proxy for any browser, system or platform*. 2018. URL: <https://www.telerik.com/fiddler>.
- [The18] The Measurement Factory. *Co-Advisor*. 2018. URL: <http://coad.measurement-factory.com/>.
- [TIB15] TIBCA Software Inc. *I/O Docs Community Edition in Node.js*. Tech. rep. 2015. URL: <https://github.com/mashery/iodocs>.
- [Uri19] P. Urien. *Remote APDU Call Secure (RACS)*. Internet-Draft. IETF, 2019. URL: <https://tools.ietf.org/html/draft-urien-core-racs-14>.

- [Ver+12] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Sam Coppens, Joaquim Gabarró Vallés, and Rik Van de Walle. *Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web*. In: *3rd International Workshop on RESTful Design (WS-REST)*. 2012. ISBN: 978-1-4503-1190-8. DOI: [10.1145/2307819.2307828](https://doi.org/10.1145/2307819.2307828). URL: <http://doi.acm.org/10.1145/2307819.2307828>.
- [Ver+14] Ruben Verborgh, Andreas Harth, Maria Maleshkova, Steffen Stadtmüller, Thomas Steiner, Mohsen Taheriyan, and Rik Van de Walle. *Survey of Semantic Description of REST APIs*. In: *REST: Advanced Research Topics and Practical Applications*. Springer New York, 2014, pp. 69–89.
- [W3C17] W3C. *HTML 5.2*. W3C Recommendation. 2017. URL: <https://www.w3.org/TR/html5/browsers.html#appcache>.
- [Wan99] J. Wang. *A Survey of Web Caching Schemes for the Internet*. In: *SIGCOMM Comput. Commun. Rev.* 29.5 (1999), pp. 36–46. ISSN: 0146-4833. DOI: [10.1145/505696.505701](https://doi.org/10.1145/505696.505701). URL: <http://doi.acm.org/10.1145/505696.505701>.
- [Wat] M. Watson. *Web Cryptography API*. Recommendation. W3C. URL: <https://www.w3.org/TR/WebCryptoAPI/>.
- [Web18] Web Polygraph. *Web Polygraph*. 2018. URL: <http://www.web-polygraph.org/>.
- [WHA18] WHATWG. *XMLHttpRequest*. Living Standard. 2018. URL: <https://xhr.spec.whatwg.org/>.
- [WSJ17] M. Wollschlaeger, T. Sauter, and J. Jasperneite. *The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0*. In: *IEEE industrial electronics magazine* 11.1 (2017), pp. 17–27.
- [Xu+08] Xiwei Xu, Liming Zhu, Yan Liu, and Mark Staples. *Resource-Oriented Business Process Modeling for Ultra-Large-Scale Systems*. In: *2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems*. Association for Computing Machinery, 2008. URL: <https://doi.org/10.1145/1370700.1370718>.
- [YM13] F. Yang and S. Manoharan. *A security analysis of the OAuth protocol*. In: *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 2013. URL: <http://doi.org/10.1109/PACRIM.2013.6625487>.

# Appendices

# Appendix A

## Cache Testing Framework

The cache testing framework is a comprehensive compliance auditing tool kit for all kinds of web caching systems. It comprises a test suite with over 415 test cases and a cache testing tool. We developed the cache testing framework for the empirical studies in Paper 4 and Paper 5. We also used this framework in combination with other tools in Paper 6 to analyze whether a tested cache or website is vulnerable to CPDoS.

After Paper 4 was accepted for the publication at the 2nd International conference on Web Studies, we published a website of our cache testing framework (see Figure A.1 <https://das.th-koeln.de/developments/cache-testing-tool/>).

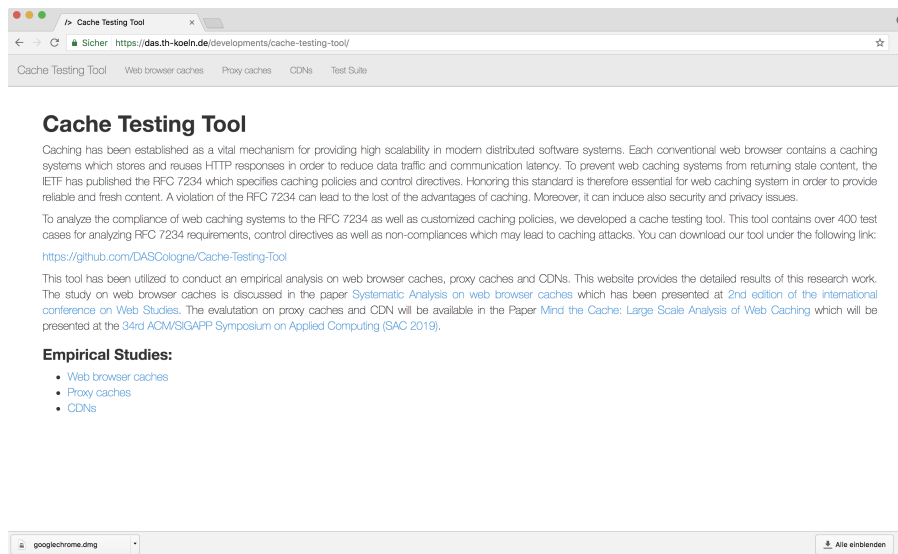


Figure A.1: Website of the cache testing framework

The website offers full access to the test suite with all test cases which can be downloaded and modified by the proposed test case definition language in Paper 4. The cache testing tool consists of two parts: a testing client and a testing server. Both can be downloaded as a standalone executable program as well as open-source Git repository via Github (Testing client: <https://github.com/das-th-koeln/Cache-Testing-Tool>; Testing server: <https://github.com/das-th-koeln/cachetestservernode>). More details on the installation process and usage of the cache testing tool can be found on the website.

# Appendix B

## CREHMA

We implemented multiple reference implementation for CREHMA in Java, JavaScript and Ruby. All implementations can be downloaded and modified as open-source software via GitHub.

### **jCREHMA**

jCREHMA is a Java implementation that is based on jREHMA. We designed CREHMA to be used in any Java-based web framework or HTTP library. For more details, please read the documentation of jCREHMA. The full source code of jCREHMA, including the documentation, can be downloaded here <https://github.com/das-th-koeln/jCREHMA>.

### **CREHMA.js**

CREHMA.js is the JavaScript adaption of CREHMA. It can be used in client-side web applications to sign XHR requests and verify HTTP responses. We use the WebCrypto API [Wat] to sign and verify the messages. For web browsers which does not support the WebCrypto API, we also provide the option to sign and verify the messages with Stanford Javascript Crypto Library (SJCL) [SHB]. Moreover, developers can use CREHMA.js in every JavaScript web framework or library such as Node.js. CREHMA.js is also freely available as open-source software on GitHub (<https://github.com/das-th-koeln/CREHMA.js>).

### **CREHMAforRuby**

The CREHMA implementation for Ruby is based on the native Net::HTTP library. For the cryptographic, we used the OpenSSL library. CREHMAforRuby can be used in any Ruby-based web library and web framework. It can be downloaded here <https://github.com/das-th-koeln/CREHMAforRuby>.