Stuttgart Media University
Faculty Print and Media

Measuring Adoption of Phishing-Resistant Authentication Methods on the Web

Master's Thesis

submitted by
Martin Bock

in fulfillment of the
requirements for the degree of
Master of Science

Matriculation Number:	40822
Course of Studies:	Computer Science and Media (M. Sc.)
First Examiner:	Prof. Walter Kriha
Second Examiner:	Benjamin Binder, M. Sc.
Date:	June 30, 2023

Ehrenwörtliche Erklärung (German)

Hiermit versichere ich, Martin Bock, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: „Measuring Adoption of Phishing-Resistant Authentication Methods on the Web“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 23 Abs. 2 Master-SPO (3 Semester) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, 30. Juni 2023 _____

Martin Bock

Abstract

Password-based authentication is widely used online, despite its numerous shortcomings, enabling attackers to take over users' accounts. Phishing-resistant Fast IDentity Online (FIDO) credentials have therefore been proposed to improve account security and authentication user experience. With the recent introduction of FIDO-based passkeys, industry-leading corporations aim to drive widespread adoption of passwordless authentication to eliminate some of the most common account takeover attacks their users are exposed to. This thesis presents the first iteration of a distributed web crawler measuring the adoption of FIDO-based authentication methods on the web to observe ongoing developments and assess the viability of the promised passwordless future. The feasibility of automatically detecting authentication methods is investigated by analyzing crawled web content. Because today's web is increasingly client-side rendered, capturing relevant data with traditional scraping methods is challenging. Thus, the traditional approach is compared to the browser-based crawling of dynamic content to optimize the detection rate. The results show that authentication method detection is possible, although there are some limitations regarding accuracy and coverage. Moreover, browser-based crawling is found to significantly increase detection rate.

Keywords — FIDO, WebAuthn, Passkeys, Adoption, Authentication, Web Crawling

Kurzfassung (German)

Passwortbasierte Authentifizierung ist trotz ihrer zahlreichen Probleme im Web weit verbreitet. Viele dieser Probleme ermöglichen es Angreifern erst, Nutzerkonten zu übernehmen. Die phishing-resistenten Fast IDentity Online (FIDO) Credentials wurden vorgeschlagen, um neue Maßstäbe für die Sicherheit und Benutzerfreundlichkeit von Authentifizierung im Web zu setzen. Mit der kürzlich erfolgten Vorstellung von FIDO-basierten Passkeys wollen branchenführende Unternehmen den großflächigen Einsatz von passwortloser Authentifizierung vorantreiben, um einige der häufigsten Angriffe zur Übernahme der Accounts ihrer Kunden zu unterbinden. Diese Arbeit stellt die erste Iteration eines verteilten Web-Crawlers vor, der die Unterstützung von FIDO-basierten Authentifizierungsmethoden im Web misst, um die Verbreitung zu beobachten und die Realisierbarkeit einer versprochenen passwortlosen Zukunft zu bewerten. Durch die Analyse der gecrawlten Inhalte wird die Machbarkeit der automatischen Erkennung von Authentifizierungsmethoden untersucht. Da das heutige Web zunehmend clientseitig gerendert wird, ist die Erfassung relevanter Daten mit herkömmlichen Scraping-Methoden eine Herausforderung. Daher wird der traditionelle Ansatz mit dem browserbasierten Crawling von dynamischen Inhalten verglichen, um die Erkennungsrate zu optimieren. Die Ergebnisse zeigen, dass die Erkennung von Authentifizierungsmethoden möglich ist, auch wenn es einige Einschränkungen in Bezug auf Genauigkeit und Abdeckung gibt. Darüber hinaus wird festgestellt, dass browserbasiertes Crawling die Erkennungsrate signifikant erhöht.

Acknowledgements

First, I would like to express my gratitude to my advisors, Prof. Walter Kriha and Benjamin Binder, for their guidance, support, and advocacy, especially when the state's research and education infrastructure provider abruptly reversed their approval of the project.

I would also like to thank my family and friends for their encouragement, support, and understanding when I couldn't spend time with them. Thank you to Stephanie Jauss for proofreading, but especially for taking on this memorable journey together and for always being there for me during stressful times.

This thesis marks the end of my studies, which would not have been possible without my parents' support. Thank you for believing in me and enabling me to pursue this path.

Contents

List of Abbreviations

2FA two-factor authentication

API application programming interface
ARKG Asynchronous Remote Key Generation
AWS Amazon Web Services

BLE Bluetooth Low Energy

CA Certificate Authority
CDN content delivery network
CMS Content Management System
COSE CBOR Object Signing and Encryption
CPU Central Processing Unit
CQL Cassandra Query Language
CrUX Chrome User Experience Report
CTAP Client to Authenticator Protocol
CWCE Cloud-based Web Crawler Engine

DBMS database management system
DHT Distributed Hash Table
DNS Domain Name System
DOM Document Object Model

eID electronic ID card

FIDO Fast IDentity Online
FIFO First-In-First-Out

GPS Global Positioning System

HIBP Have I Been Pwned
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service
IAM Identity and Access Management
IANA Internet Assigned Numbers Authority
IDN Internationalized Domain Name
IP Internet Protocol
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
ISP Internet Service Provider

JMX Java Management Extensions
JS JavaScript
JVM Java Virtual Machine

MAC message authentication code
MFA multi-factor authentication
MITM Man-In-The-Middle
mTAN mobile Transaction Authentication Number

NAT Network Address Translation
NFC Near-Field Communication
NIST National Institute of Standards and Technology

OS operating system
OTP one-time password

PIN personal identification number

RAM Random Access Memory
Regex regular expression
RP Relying Party
RSA Rivest–Shamir–Adleman

SEO Search Engine Optimization
SIM Subscriber Identity Module
SMS Short Message Service
SoC System on a Chip
SPA Single-Page Application
SQL Structured Query Language
SSH Secure Shell
SSO Single Sign-On

TCP Transmission Control Protocol
TLD Top Level Domain
TLS Transport Layer Security
TOTP Time-based One-Time Password
TTFB Time to First Byte

U2F Universal 2nd Factor
UDP User Datagram Protocol
UI user interface
ULA Unique Local Address
URI Uniform Resource Identifier
URL Uniform Resource Locator
USB Universal Serial Bus
UX user experience

vCPU virtual Central Processing Unit
VM Virtual Machine
VPN Virtual Private Network

W3C World Wide Web Consortium
WebAuthn Web Authentication
WWW World Wide Web

Zstd Zstandard

List of Tables

List of Figures

List of Listings

1. Introduction

More and more things in everyday life are taking place online. From social media to online shopping and online banking to digital government services, many activities require some form of a person's identity. People verify their identity by authenticating themselves to the service provider using personal credentials. Typically, this means a combination of a publicly-known username and a confidential password that only legitimate users should know.

Password-based authentication has many flaws that malicious actors are exploiting continuously. For instance, users are expected to remember their passwords but to prevent others from being able to guess them, unique passwords must be long and complex. However, humans have a hard time memorizing a long sequence of random, incoherent characters.

Because passwords are shared secrets, users are supposed to hand their passwords over to the service provider for authentication. At the same time, users do not have any way of influencing or even knowing how service providers handle their secrets. Time and time again, *credential spills* are surfacing on the Internet because a service provider did not take appropriate measures to prevent leaking their users' passwords. And since secure passwords should be long and complex, users tend to reuse them. In fact, research suggests that 70 % of people reuse their passwords for some, most, or all of their online accounts [192]. As a result of credential spills, these passwords fall into the hands of attackers that can use them for *credential stuffing* attacks, trying to sign in on other online services with a known username or email and password combination.

However, another heavy burden is placed on users: They are expected to verify that the service provider asking for their password is genuine and that they are not being tricked into handing their credentials over to an attacker whenever they authenticate anywhere. Research shows that users have no viable way to avoid getting phished if attackers use appropriate techniques and catch the right moment [6, 89, 180].

To solve these problems once and for all, a new authentication technology emerged: Fast IDentity Online (FIDO) credentials. Industry leaders have joined forces to form the FIDO *Alliance*, which is set out to protect users from credential spills, phishing, and other password-related threats. FIDO credentials are based on asymmetric cryptography to avoid sending confidential data over the wire, which makes credential spills impossible. Moreover, because FIDO credentials are cryptographically bound to the associated service provider's domain, phishing is rendered impossible, releasing users from the burden of verifying their counterparts for every authentication operation.

Since their inception, the adoption of FIDO credentials has gradually increased as part of multi-factor authentication (MFA) flows used by a small group of technically well-versed people. That is because the most secure form of a FIDO credential, a physical security key, involves additional cost and has usability downsides to a point where ordinary users do not realize the importance of the provided security benefits [65, 124]. To lift FIDO credentials out of their niche existence and push their widespread adoption, *Apple*, *Google*, and *Microsoft* have collaborated with other members of the FIDO *Alliance* in a joint effort to replace password authentication with *passkeys*, i.e., *multi-device FIDO credentials*, that aim at mitigating user experience (UX) issues that previously hindered large-scale adoption.

To comprehend the viability of a *passwordless* future, research needs to quantify support for FIDO credentials on the web so that the adoption rate can be monitored over time. With that goal in mind, this thesis presents the first iteration of a distributed crawler to measure the adoption of phishing-resistant authentication methods on the web.

Because content on the web is increasingly rendered on the client side, it is questionable whether traditional crawling of static web content is able to detect the use of the relevant Web Authentication (WebAuthn) browser application programming interface (API) that facilitates the communication with FIDO authenticators on the client side. To investigate this, every targeted website is crawled using a traditional Hypertext Transfer Protocol (HTTP) client that fetches static content and, in addition, using a dynamic crawler that remote-controls a full web browser and can collect client-side rendered content.

Therefore, this thesis seeks to answer the following research questions:

R1 Is it possible to automatically detect usage of authentication methods on the web with a reasonable accuracy?

R2 Does the detection rate differ for static and dynamic web crawling?

The remainder of this thesis is structured as follows: In section 2, previous work related to authentication on the web and building distributed web crawlers is reviewed. Section 3 establishes requirements and presents the software architecture for the proposed crawler. Section 4 describes the crawler's implementation, while section 5 covers the automated deployment process and some infrastructural optimizations. The crawl results are analyzed quantitatively and qualitatively in section 6. Finally, section 7 discusses the limitations of the presented approach and gives an outlook on possible improvements for future work before section 8 concludes the thesis.

2. Related Work

This thesis is based on two distinct categories of previous research. On the one hand, knowledge of the available means by which a user can be authenticated on the Internet is the foundation for evaluating and comparing characteristics such as their usability and possible attack surfaces. This evaluation motivates exploring the adoption of superior authentication methods across the web. Related research is reviewed in section 2.1.

On the other hand, looking at research on the design of distributed web crawlers and learning from previous findings is essential to carry out large-scale measurements on the web. Section 2.2 gives an outlook of previous work in this area.

2.1. Means of Authentication

Many use cases on the World Wide Web (WWW) are only viable if users can uniquely and securely authenticate themselves to service providers. This includes all services for which users pay money as a purchase or subscription. This section discusses several means of authentication used on the web, including their benefits and drawbacks. First, section 2.1.1 discusses passwords and their risks, while section 2.1.2 introduces several subsequent MFA mechanisms. Next, section 2.1.3 introduces the essentials of FIDO authentication. Section 2.1.4 then discusses obstacles in widespread FIDO adoption, while section 2.1.5 emphasizes its phishing resistance. Finally, in sections 2.1.6 and 2.1.7, the newest developments of FIDO-enabled authentication are presented.

2.1.1. Password-Based Authentication and Its Risks

Traditional password-based authentication has many shortcomings: Passwords that are easy to remember are also easy to guess, while secure passwords are hard to remember. Passwords are prone to be phished or spilled by service providers who negligently handle them. Indeed, the number of reported credential spill incidents nearly doubled from 2016 to 2020[1], even though the number of leaked credentials decreased [185]. Furthermore, because remembering (secure) passwords is challenging, many users tend to re-use them [192]. In 2022, phishing and personal data breaches were the two top-reported Internet-related crimes worldwide [95]. Both are primarily enabled by the inherent risks of using password-based authentication.

Complexity vs. Length

How well a password can protect an account is usually determined by its uniqueness. To make a password unique, it should be randomly generated. In order to describe the level of randomness, an entropy value in bits can be calculated. Entropy specifies the level of uncertainty in a variable [171]. Equation (1) shows that entropy E in bit is calculated based on the number of possible symbols N (complexity) to the power of the number of symbols L (length) to the $\log_2$.

$$E = \log_2(N^L) \tag{1}$$

To prevent accounts from being compromised, for many years, the National Institute of Standards and Technology (NIST) and similar institutions recommended implementers to impose

[1] 2016: 52 incidents (avg. credentials spilled 3.3 B), 2020: 117 incidents (avg. credentials spilled 1.8 B)

a set of rules on user-chosen passwords to ensure that an adequate level of complexity was present [39]. However, these rules actively harm usability because passwords that fulfill complex character set requirements are not easy to remember. Additionally, eq. (1) makes it fairly obvious that increasing the password length is a far more effective way to increase entropy compared to increasing the character pool size. In light of these reasons, the prevailing opinion has shifted and NIST now advises against using any complexity requirements other than a minimum password length [81]. Using longer *passphrases* composed of multiple words instead of traditional passwords is also easier to remember and type [174].

Password-Based Attacks

Password entropy is an essential criterion for account security. However, table 1 shows that the strength of a password is irrelevant for many common attacks [189]. Thus, many service providers try to encourage users to protect their accounts from takeovers by using additional or alternative authentication methods. Not all of these authentication methods provide the same level of protection, though. A subset of approaches and their respective shortcomings is detailed in the following.

Attack	Description	Attacker Has Exact Password
Brute Force	Testing a list of passwords against a single account. Passwords could be generated or taken from a dictionary.	×
Credential Stuffing	Testing credentials obtained from a data breach on other accounts of the same user.	✓
Password Spraying	Testing a single weak password against multiple accounts.	×
Phishing	Tricking a user into handing over their password.	✓
Keystroke Logging	Intercepting a user's input through malware.	✓
Local Discovery	Discovering a password physically or digitally, e.g., on a paper note or in a text file on a computer.	✓

Table 1: Common Authentication Attacks [134, 143, 152, 189]

Password Reuse

In 2018, a representative survey found that 70 % of adults in the United States reuse their passwords for some, most or all of their accounts. In contrast, only 22 % said that they used unique passwords [192]. To protect users from brute force attacks like password spraying, NIST advises implementers to ensure passwords do not contain "commonly used, expected or compromised values" and recommends to verify that passwords do not appear in breach corpora[2] [81]. Have I Been Pwned (HIBP) is a well-known, publicly available breach corpus whose API served up to 1.26 billion requests per day in 2021. The corpus includes around 847 million breached passwords [90, 91]. Among others, the password manager *1Password* uses HIBP to ensure password

[2] lists of previously compromised passwords

uniqueness [5]. To avoid exposing a user's passwords to HIBP, their password query API design takes advantage of the mathematical property *k-Anonymity* [8, 90]. For instance, hospitals that release patient data and want to avoid disclosing personal information also make use of *k-Anonymity* [161]. In essence, *k-Anonymity* is provided if a dataset has k identical records for every *quasi-identifier*, which is some form of distinctive data [161]. HIBP achieves *k-Anonymity* by splitting a password hash into prefix and suffix. The API accepts queries for a fixed-size prefix. Comparing the returned suffixes is done locally by the client. The used hash functions provide two important properties: Firstly, since small changes in a hash function's input result in a significantly different output, one cannot infer the contents of one hash from another. This algorithmic property is called the *avalanche effect* [67]. Secondly, since hashes are reasonably uniformly distributed, there is roughly the same number of returned suffixes for each given prefix [8].

Using a breach corpus to reject previously breached passwords can be an effective measure to prevent password spraying and may also help with other brute force attacks like credential stuffing [143, 152, 185]. However, effectively blocking 847 million password choices negatively impacts usability. Rejecting this number of possible passwords is prohibitive for many users. A 2021 survey showed that 52 % of people memorize their passwords. Only 24 % use a dedicated password manager [28]. Moreover, while rejecting passwords that appear in breach corpora does protect users to some extent, it cannot fully prevent users from reusing their passwords on multiple websites, which still leaves them vulnerable to credential stuffing and phishing attacks.

2.1.2. Multi-Factor Authentication

Section 2.1.1 shows how risky using passwords as a single factor of authentication can be. In fact, research from Microsoft suggests that a non-randomly-generated password's length and complexity mostly do not matter in case of a breach, while MFA would have stopped 99.9 % of analyzed account compromises [189].

Over the last few years, surveys have found a continual increase in MFA usage from 28 % in 2017 to 79 % in 2021 in the United States and United Kingdom [11, 44].

The basic concept of MFA is to combine different types of evidence confirming a user's identity. Table 2 shows the four possible factors with some concrete examples. While requiring multiple inputs of a single factor may provide some security benefits, it cannot be considered true MFA [128]. The remainder of this section focuses on several common MFA methods and their drawbacks. Additionally, section 2.1.3 describes FIDO authenticators, which may also be used in a MFA context.

Factor	Examples
Knowledge	Password, personal identification number (PIN), security question
Possession	Software certificate (e.g., Secure Shell (SSH) private key), physical security key, Subscriber Identity Module (SIM) card
Inherence	Fingerprint, facial recognition, iris scan
Location	Internet Protocol (IP) address, Global Positioning System (GPS) coordinates

Table 2: Authentication Factors [128]

SMS-Based One-Time Passwords

In 2021, receiving one-time passwords (OTPs) via Short Message Service (SMS) [1] was the most commonly used MFA factor with 85.2 % in the United States and United Kingdom [44]. To use this method, users provide their cellphone number on activation and subsequently receive their OTPs over the cellular network. The method is also widely used in the banking sector, where the OTPs are mostly referred to as mobile Transaction Authentication Numbers (mTANs). One reason SMS-based OTPs are so widely used may be that nearly every user already has a cellphone and can receive SMS messages [176]. Moreover, the concept of receiving six or eight numbers in an SMS message and copying them into a login form is easily grasped. Therefore, it may be less intimidating than other described methods. With a password being the first authentication factor (proof of knowledge), the main idea is that in order to receive the SMS OTP, a user has to prove possession of the cellphone tied to the registered cellphone number as a second factor.

However, many ways exist to circumvent SMS-based MFA. Since SMS messages are not end-to-end encrypted, the attack surface expands to three or four involved parties where the OTP may leak.

One way for an attacker to intercept OTPs is to perform a Subscriber Identity Module (SIM) swapping attack where the telecommunication provider is tricked into sending a new SIM card associated with the victim's cellphone number to the attacker. This can be exceptionally easy as some phone carriers verify one's identity merely by asking for the full name, physical address, and date of birth – all of which can be public knowledge [70, 159, 179].

Another method an attacker could use is wireless interception, although the effort involved is probably feasible only for high-value targets. After jamming 4G and 5G cellular frequencies and forcing a targeted device to downgrade to 3G or GSM, an attacker could record and decrypt the relevant packets containing the target OTP [76, 139].

Intercepting the OTP on the receiving phone itself through smartphone trojans, on the other hand, is a method that scales significantly better. Trojans like *CruseWind* are aimed specifically at intercepting SMS OTPs [64]. Research has also repeatedly shown that intercepting mTANs sent by financial institutions using smartphone trojans is possible [54, 168].

Perhaps the most obvious way of obtaining an SMS-based OTP, though, is just asking the victim for it using a real-time phishing attack. Siadati et al. found that 50 % of users can be tricked into forwarding an OTP with a phishing message sent via SMS shortly after the attacker caused an OTP SMS to get delivered to the victim's phone. For example, "Did you request a password reset for your Gmail account? Delete this message if you did. Otherwise, send 'Cancel' + the verification code we just sent to you" was identified to be the most effective of all tested phishing messages [172].

In conclusion, SMS-based MFA is the weakest of all showcased methods, and using it should be avoided due to the large attack surface it provides. Still, 53.4 % of respondents in a 2021 survey said that they would adopt SMS MFA for a new account [44]. This may either indicate that people are unaware of the dangers of using SMS OTPs, or that the ease of use outweighs the perceived risk.

Time-Based One-Time Passwords

Time-based One-Time Passwords (TOTPs) are generated using the *HMAC-SHA-1* algorithm [112]. A time step value based on the current *UNIX* timestamp [178] is used as the algorithm's input, along with a shared secret [126].

Let K be a static symmetric key shared between the server and the client. Let T_0 be the initial timestamp (defaults to 0), T_c the current timestamp, and X the time step size (defaults to 30 seconds) [126]. Then

$$\begin{aligned} T &= \frac{T_c - T_0}{X} \text{ and} \\ \text{TOTP} &= \mathit{truncate}\Big(\mathit{hmac}(K, T)\Big) \ . \end{aligned} \tag{2}$$

Because users need to manually enter the rolling result into a login form, the value returned by the *HMAC* function is truncated to a 4-byte string [126]. In order for the website provider and the user to be able to generate the same TOTP at the same time, their local clocks need to be synchronized. However, clock skew may cause one of the local clocks to get slightly out of sync. Moreover, networks may introduce a varying degree of latency, so TOTPs can arrive one or more time steps later at the authenticating server. For those reasons, RFC 6238 recommends that implementers define an acceptable window of past and future time steps to improve usability [126].

In the previously mentioned survey, using authenticator apps was the third-most-popular second factor with 44.4 % of respondents [44]. Most apps generate TOTPs using a key Uniform Resource Identifier (URI) format initially defined by Google [80]. To make importing easier, most apps support scanning a *QR code* containing a URI similar to listing 1.

Listing 1: OTP URI Example

```
1 otpauth://totp/ACME%20Corp:Jane%20Doe?secret=7LAY5SF7XPBZKR9HVSLEBPSD3SV8G8JF&issuer=
    ↪ACME%20Corp
```

Unfortunately, in the past, Google's reference server implementation only used a default key length of 80 bit [82]. This is in violation with RFC 4226, which defines a minimum key length of 128 bit and recommends using 160 bit [125]. Research shows that using a shorter key length makes recovery of the key feasible if any two OTP values are known [184]. Regrettably, many websites still use an insecure key length. In a small, biased sample, the following vendors were found to have issued 80 bit keys over the last few years: Autodesk, Discord, GitHub, Microsoft, Nextcloud, PayPal, Tumblr, Twitter, and Ubiquiti. It is important to note that it has not been verified whether these companies are still issuing new keys of this length today.

Aside from individual implementation issues, using TOTPs can significantly improve account security. However, even accounts secured with TOTP MFA are susceptible to account takeovers using a phishing attack. After all, humans are still ultimately responsible to verify that they enter their credentials only on the correct website domain. And even though password managers can help validate the domain based on auto-filling the TOTP on login forms, many people are used to manually entering the credentials when auto-filling does not work correctly. There are countless ways an attacker can trick victims into visiting phishing sites, for example, utilizing Internationalized Domain Names (IDNs) that are visually indistinguishable from the real domain [89]. Since TOTPs are replayable, they are also susceptible to real-time phishing attacks [6]. Humans are always the weakest link and should not be responsible for validation

tasks that are extremely difficult for people but relatively easy for computers. Adoption of FIDO-based authentication covered in section 2.1.3 has the potential to free users from this burden.

Some non-TOTP MFA apps like *Octa Verify* [141] or *Duo Mobile* [46] are used in a similar fashion, but login attempts trigger a push notification. In the app, users may then permit or deny those attempts. Apart from being vulnerable to phishing attacks, push-based MFA is also vulnerable to *MFA fatigue attacks*, where an attacker repeatedly triggers a push notification, hoping the user gets annoyed until they finally authorize the login attempt. This approach has recently enjoyed particular popularity [2, 40]. For instance, Uber was successfully breached using an *MFA fatigue attack* in 2022 [182].

2.1.3. FIDO2 and WebAuthn Essentials

In collaboration with the World Wide Web Consortium (W3C), the *FIDO Alliance* published the protocol standards WebAuthn and Client to Authenticator Protocol (CTAP) for a novel, easy-to-use authentication mechanism based on asymmetrical cryptographic key pairs. The project is commonly known as *FIDO2* [34, 100]. At the time of writing, over 95 % of global web browsers already support the WebAuthn API [56]. Recently, news outlets reported on an industry-wide push to replace passwords with *passkeys*, which are *FIDO* credentials that are stored on and synchronized between a user's devices [38, 51, 71, 77, 108, 113]. Section 2.1.6 presents the underlying technical concept of multi-device FIDO credentials.

Terminology

Before addressing how FIDO authentication works in detail, some relevant terminology needs to be introduced.

FIDO Alliance The *Fast IDentity Online (FIDO) Alliance* is an organization made up of various stakeholders within the information technology industry. Its 41 board members include 1Password, Amazon, Apple, Google, Intel, LastPass, Lenovo, Mastercard, Meta, Microsoft, PayPal, Samsung, Visa, Yubico, and others [68]. Together with its 81 sponsor level, 9 government level, and 188 associate level members, there are a total of 319 members [68].

FIDO2 *FIDO2* is the name of the authentication framework. It is a joint project of W3C and the *FIDO Alliance*. At its core are the WebAuthn API specified by W3C and CTAP, which is specified by the *FIDO Alliance* [32, 34].

WebAuthn The Web Authentication (WebAuthn) API was standardized by W3C in 2019 [100]. It is a JavaScript (JS)-based browser API that exposes a *Credentials* interface that can be used to interact with FIDO authenticators to register and authenticate using FIDO credentials [32].

CTAP The Client to Authenticator Protocol (CTAP) defines communication with FIDO authenticators. An authenticator could be accessed over a transport method such as Universal Serial Bus (USB), Near-Field Communication (NFC), or Bluetooth Low Energy (BLE) [34]. The second version of the standard, *CTAP2*, introduces CBOR Object Signing and Encryption (COSE) [167], protocol extensions, and new attestation formats [34]. It is backward compatible

with the first protocol version, *CTAP1*, which is primarily known as Universal 2nd Factor (U2F) [34]. Modern, *FIDO2*-compatible authenticators communicate using *CTAP2*.

Platform Authenticator A platform authenticator is attached to a client device and is usually not removable from the device. For example, this could be a fingerprint sensor on a smartphone with a secure element where key material is stored. The WebAuthn specification classifies the enumerated `AuthenticatorTransport` of platform authenticators as `internal` [32].

Roaming Authenticator In contrast to platform authenticators, roaming authenticators are attached using cross-platform transports like USB, NFC, or BLE and can be removed from the client device. For example, this could be a physical security key. A roaming authenticator sets the enumerated `AuthenticatorTransport` value according to its transport method, for example, `usb`. An authenticator may support more than one transport method [32].

Relying Party Services that want to authenticate a user with *FIDO2* are considered a Relying Party (RP). RPs may explicitly define an *RP ID* based on their domain name, excluding scheme, port, and path. If a subdomain is part of the *ID*, any lower-level domain is also valid, but not vice versa. For example, valid *IDs* for a website with implicit *ID* served at `https://auth.example.com/login` would be `auth.example.com` and `example.com`, but not `v2.auth.example.com` or `example.org`. A FIDO credential is cryptographically bound to an *RP ID*, so the *ID* needs to be considered immutable [32]. Section 2.1.5 details the importance of the *RP ID* for phishing-resistance.

Attestation Some RPs may need additional assurances that an authenticator complies with legal or security requirements. To do this, RPs may specify an attestation conveyance preference. This is particularly relevant in an enterprise context, where RPs may want to identify individual authenticators in a controlled deployment (`enterprise`). For instance, attestation may also be used to prove an authenticator has a particular certification (`direct`). Because attestation statements have the potential to enable user tracking across the web, the `indirect` option allows authenticators to replace authenticator-generated attestation statements with ones generated by an *Anonymization Certificate Authority (CA)*. Most RPs, however, do not need such assurances, so they should use the `none` option [32].

User Verification RPs may require or prefer the authenticator to verify the user before being able to use a FIDO credential. The credential must then be unlocked, for example, using biometrics or a personal identification number (PIN). In the returned response, an authenticator indicates whether it verified the user by setting a `UV` flag. By default, user verification is preferred but not required [32].

Discoverable Credentials Client-side discoverable FIDO credentials are *discoverable* for clients without the RP having to provide any allowed *credential IDs* first. This means that previous RPs do not necessarily need to identify the user beforehand [32]. Usually, *platform authenticators* have the ability to store discoverable credentials, while *roaming authenticators* may not, for example, because of limited storage space. Discoverable credentials were previously referred to as *resident credentials* or *resident keys* [32].

Cryptographic Algorithm Identifiers Authenticators can support a number of cryptographic algorithms. These algorithms are identified using a unique integer value defined in the *Internet Assigned Numbers Authority (IANA) COSE Algorithms Registry* [92]. RPs specify their supported algorithms when using the WebAuthn credentials interface in the `pubKeyCredParams` array [32]. For example, support for `ES256`[3], which is based on elliptic curve cryptography, and `RS256`[4], which is based on the Rivest–Shamir–Adleman (RSA) cryptosystem, would result in a value of `[-7, -257]` [92].

Basic Functionality

The basic idea of FIDO is to replace knowledge-based authentication using a shared secret that may be memorized (password) with a possession-based asymmetric cryptographic key pair. By using a signature-based authentication process, users do not need to send a secret to the authenticating service to prove their identity. Instead, the RP sends a challenge that the user's authenticator signs or encrypts using its private key as proof of possession. This way, the user's secret is never transmitted to the authenticating service, making Man-In-The-Middle (MITM) eavesdropping attacks impossible. The user's secret cannot even get exposed if a data breach occurs at the authenticating service[5] – because the service only ever stores and has access to the user's public key and the unique credential ID [32].

The private key never leaves a roaming authenticator like a physical security key. This makes it practically impossible for an attacker to extract the private key, even if the user's computer is compromised. Some security keys do not store the private key directly but derive it on demand from the *RP ID*, a random nonce, and a device-specific secret using a message authentication code (MAC) function [138]. This approach eliminates any storage capacity limitations, which means the security key can be used on an infinite number of websites.

FIDO2 can be used as a singular factor[6] or in combination with other factors in the context of MFA [32]. RPs can require the authenticator to verify the user through biometrics, a PIN, or other means [32]. User verification is especially useful when authenticating users with a FIDO credential as a single factor. One can argue that the combination of possessing the authenticator and knowing the PIN or presenting one's face or fingerprint already constitutes MFA.

Registration Ceremony

Before an authenticator can be used, it has to create a new credential for the RP during a registration ceremony. The ceremony is initiated through a `navigator.credentials.create()` browser API call like to the one in listing 2. Beforehand, the client-side code fetches the *RP ID*, a random challenge, and other creation options from the server. For instance, the RP may exclude credentials that the user has already registered. When the WebAuthn API is invoked, the browser ensures that the website's origin matches the *RP ID* and that the origin's scheme is `https` [32].

When the browser passes the request to the authenticator using CTAP, the authenticator typically ensures user presence – for example, by requiring the user to press a physical button.

[3] ECDSA with SHA-256
[4] RSASSA-PKCS1-v1 5 with SHA-256
[5] also known as credential spilling
[6] commonly referred to as "passwordless"

The authenticator then generates an asymmetric key pair with the *RP ID* as an input parameter. It may use a one-way keyed function like `HMAC-SHA256` with the *RP ID* and a random nonce as input and a device-specific secret generated on-chip during manufacturing as the key. The output of that function becomes the private key. The nonce value, in combination with a MAC, becomes the *credential ID*[7]. This ID is a unique identifier for the corresponding FIDO credential [32, 34, 138].

The returned result of the WebAuthn API call contains the *credential ID*, a signed challenge response, and the corresponding public key within an attestation object. The WebAuthn specification requires RPs to follow a 27-step procedure when registering a new FIDO credential. Among other things, implementers must validate the credential origin, the signed challenge response, the correct response type, and optionally, whether the authenticator verified the user's presence. To validate the challenge response, the RP verifies that the public key is consistent with the challenge response signature. After successful validation, the RP can store the *credential ID*, the public key, and optionally, detailed information on the authenticator, like available transport methods [32].

Listing 2: Creating FIDO Credentials

```
const credential = await navigator.credentials.create({
  publicKey: {
    // Relying Party
    rp: {
      name: "ACME Corp",
      id: "auth.example.com", // (optional)
    },

    // User data
    user: {
      id: Uint8Array.from("ABC123", c => c.charCodeAt(0)),
      name: "jane.doe@example.com",
      displayName: "Jane Doe",
    },

    // Challenge generated by the server
    challenge: Uint8Array.from(
      randomStringFromServer, c => c.charCodeAt(0)),

    // Accepts ES256 or RS256 credentials, but prefers ES256
    pubKeyCredParams: [
      {
        type: "public-key",
        alg: -7, // ES256
      },
      {
        type: "public-key",
        alg: -257, // RS256
      },
    ],

    // Relying Party does not care about attestation (optional)
    attestation: "none",

```

[7] equivalent to a U2F *key handle*

```
    // Exclude previously registered authenticators (optional)
    excludeCredentials: [
      {
        type: "public-key",
        id: Uint8Array.from(window.atob(
          "NjFjZmQ5MDliZDU4MTVjYjVjZmY3NzU3YThiNzlkY2UK"),
          c => c.charCodeAt(0)
        ),
      }
    ],
  },
});

console.log(credential);

// PublicKeyCredential {
//   type: "public-key",
//   id: "OWFiZGIwYjYxZTJkNzExZTFjZDRlYzcxYTJmNGNlNGYK",
//   rawId: ArrayBuffer(44),
//   response: AuthenticatorAttestationResponse {
//     clientDataJSON: ArrayBuffer(121),
//     attestationObject: ArrayBuffer(306),
//   },
// }
```

Authentication Ceremony

To authenticate a user, RPs use the same challenge-response pattern. The authentication ceremony is initiated by a `navigator.credentials.get()` call similar to the one in listing 3. Of course, RPs should verify that the client supports WebAuthn. Testing whether the `window.PublicKeyCredential` property is present is one way to ensure support (lines 1-3). In preparation for authenticating users, the RP needs to fetch a new random challenge and optionally the *RP ID* from the server. For instance, if WebAuthn is used in a MFA context, fetching the user's registered FIDO credentials and allowing only those may also make sense. When invoking the WebAuthn API, the browser checks that the website's origin matches the *RP ID*'s scope and that the scheme is `https`. The authenticator then retrieves or generates the private key corresponding to the *RP ID*, and possibly one of the allowed *credential IDs*. It uses the private key to sign the server's challenge and returns the response along with the utilized credential's ID. The RP then, among other things, verifies that the signature matches the public key on file for the *credential ID* in question and that the origin in the returned client data object is – in fact – the RP's origin. If all verifications are successful, the user can be signed in [32, 151].

Listing 3: Authenticating using FIDO Credentials

```
if (!window.PublicKeyCredential) {
  // Client has no WebAuthn support
  // Handle error...
}

const assertion = await navigator.credentials.get({
  // Challenge generated by the server
  challenge: Uint8Array.from(
    randomStringFromServer, c => c.charCodeAt(0)),
```

```
10
11    // Relying Party ID (optional)
12    rpId: "auth.example.com",
13
14    // Allow registered credentials (optional, useful for MFA)
15    allowCredentials: [{
16      id: Uint8Array.from(
17        "OWFiZGIwYjYxZTJkNzExZTFjZDRlYzcxYTJmNGNlNGYK",
18        c => c.charCodeAt(0),
19      ),
20      type: 'public-key',
21      transports: ['usb', 'ble', 'nfc'],
22    }],
23  });
24
25  console.log(assertion);
26
27  // PublicKeyCredential {
28  //   type: "public-key",
29  //   id: "OWFiZGIwYjYxZTJkNzExZTFjZDRlYzcxYTJmNGNlNGYK",
30  //   rawId: ArrayBuffer(44),
31  //     response: AuthenticatorAssertionResponse {
32  //     authenticatorData: ArrayBuffer(191),
33  //     clientDataJSON: ArrayBuffer(118),
34  //     signature: ArrayBuffer(70),
35  //     userHandle: ArrayBuffer(10),
36  //   },
37  // }
```

2.1.4. Obstacles of FIDO2 Adoption

Two aspects are decisive for the widespread adoption of *FIDO2*: Service providers need to support it – and users need to use it. On the user side, a good UX will determine whether people will use FIDO over traditional authentication methods.

In 2020, Lyastani et al. carried out the first large-scale usability lab study about *passwordless* authentication. While it is encouraging that results show users are willing to accept a security key as a direct replacement for passwords for single-factor authentication, they also identified users' concerns that may hinder widespread FIDO adoption [124]. For instance, participants were afraid someone could gain access to their accounts with a lost or stolen security key [124]. Several participants raised the question of how to "revoke" or "recover" an account's access. They also voiced their desire for a backup authentication method [124]. However, a participant, who claimed to have been a victim of password theft in the past, also pointed out that the disappearance of a security key from one's possession immediately warns the user of a potential account intrusion attempt. In comparison, passwords cannot offer this implicit guarantee that no one else can access one's account as long as one is in possession of their security key [124]. Lyastani et al. also found that even after watching a video-based introduction to the technology, participants only had a rudimentary mental model of the technology since obvious misconceptions were present in 59 % of free-text responses. For example, participants questioned whether it was possible to track their exact location once they inserted their security key [124]. Lyastani et al. conclude that a lack of technical background knowledge and the associated lack of trust can be one of the biggest hurdles in widespread FIDO adoption [124].

Keil et al. found similar trust issues due to a lack of technical knowledge in their qualitative user study. They examined users' impressions of usability, their perception of security, and overall acceptance of using a German electronic ID card (eID) as a second-factor FIDO authenticator. Although the login was generally perceived to be easy, non-tech-savvy participants were found to struggle with the setup [104]. Additionally, misconceptions regarding the transmission of personal information to the authenticating service were observed. 45 % of participants[8] thought that at least some personal information would be shared with service providers even though that was not the case [104]. In fact, the app[9] released by the German government for interacting with the eID shows precisely what information will be read from the eID's chip before entering the PIN. In the case of the study prototype, the app displayed that only a "pseudonym" would be read [104].

Farke et al. accompanied a small software company deploying *FIDO2* security keys as a single authentication factor in a qualitative study. Most participants considered FIDO-based authentication usable, but several stopped using the security key as it was slower than using the password manager built into their web browsers. Moreover, the security benefits of *FIDO2* were mostly intangible and participants perceived them as unnecessary [65]. However, participants' statements suggested that the habit of using passwords were deeply ingrained and could not be easily replaced within the study's timeline of just four weeks [65].

Lassak et al. conducted three studies focusing on smartphones as FIDO authenticators, as they have the potential of a more widespread adoption compared to specialized security hardware that is entailed with extra costs. In the first study, they established a baseline of common misconceptions when using smartphones as authenticators [115]. In the second study, they educated participants on the technology behind FIDO and used a co-design approach to formulate useful notifications to be used in a WebAuthn authentication flow. This approach aims to benefit from end-users' creativity and opinions [115]. Finally, the third study tested the effectiveness of eliminating misconceptions of the co-designed browser notifications on a new group of participants [115]. Probably due to misconceptions that were initially spread when biometric phone unlocking was first introduced and the superficial appearance of users signing into web services only with their fingerprint or face, 67 % of participants in the first study incorrectly assumed their biometrics were sent to the authenticating website [115]. Another common misconception was that the fallback mechanism to unlock one's phone by PIN, pattern, or password was not applicable in a FIDO context [115]. Although the co-designed notifications were able to partially address misconceptions in the third study, and most participants indicated they were willing to adopt biometric FIDO authentication, many participants still held key misconceptions. Particularly, people were still confused about where biometric information was stored. Lassak et al. concluded that there was a need for more expansive education efforts [115].

In 2021, when the studies by Lassak et al. were conducted, participants also incorrectly assumed that they could use their other devices to sign in to websites once they registered one of them. At the time, FIDO authenticator implementations in iOS and Android were not designed to transfer private keys from one device to another. However, this constraint has been addressed by the *FIDO Alliance* with the introduction of *multi-device FIDO credentials*, which are described in section 2.1.6.

Similarly to their research in 2020, Farke et al. accompanied the introduction of Microsoft's platform authenticator *Windows Hello* in a small business through a qualitative study in 2022. In contrast to the study in 2020, participants perceived *Windows Hello* to be faster and more

[8] absolute: 9 of 20 participants
[9] AusweisApp2

responsive than the traditional Windows login process [66]. Although they liked using the FIDO-based authentication, participants tended to use PINs as a replacement for their longer passwords instead of using biometrics [66].

Research has shown that one major issue with FIDO authenticators like security keys is the risk of token loss and the means of token recovery and revocation. Schwarz et al. tried to explicitly solve the token loss issue by designing a cloud-based credential service acting as a FIDO authenticator that generates recoverable FIDO credentials on demand using immutable data like name, date of birth, and place of birth from government-issued eIDs [170]. The idea is that this data will not change, even when a refreshed eID is issued [170]. Additionally, there is no added cost for users since they already possess an eID [170]. The downside of this system design, though, is that the web service has to be trusted, and it must guarantee high availability. Previously cited research has repeatedly shown a lack of users' trust, especially among people who are not technically inclined. These findings raise significant doubts about the success of such a design.

In the field of FIDO token revocation, only little published research is available. However, Hanzlik et al. recently released a preprint version of their yet-unpublished paper[10] proposing a revocation procedure based on *BIP32* key derivation, mainly used in cryptocurrency wallets.

Regarding token recovery, there have been some developments in recent years. For example, with roaming authenticators in mind, security key vendor Yubico proposed a WebAuthn protocol extension to create a backup authenticator that shares key material with the primary authenticator. It is founded on a new cryptographic primitive called Asynchronous Remote Key Generation (ARKG) [121]. Frymann et al., in cooperation with Yubico, published a cryptographic analysis of the proposal proving its security [72]. Concerning platform authenticators, in 2022, the *FIDO Alliance* introduced the concept of multi-device FIDO authenticators, commonly referred to as passkeys. They will be discussed in section 2.1.6.

While UX issues are a potential major hurdle in widespread FIDO adoption, a large number of service providers need to integrate WebAuthn support into their offerings in order for general user adoption to even become possible. Alam et al. in 2019 concluded that comprehensive information on secure implementations of WebAuthn was extremely sparse [7]. Most implementers simply do not read the W3C specification, and apart from that, the web is full of unfinished, broken, or unmaintained code examples and libraries [3, 4, 63]. After analyzing implementers' questions on developer forums like *Stack Overflow*, Alam et al. identified a need for official, more comprehensive educational materials [7]. Fortunately, the situation has been improving over recent years, with more documentation by large vendors and open-source plug-and-play WebAuthn libraries becoming available since the introduction of multi-device FIDO authenticators and the adoption of the technology by industry leaders like Apple and Google [14, 16, 22, 24, 114, 193].

2.1.5. Towards Phishing-Resistance

Phishing was by far the most reported cyber crime category worldwide in 2022 [95], emphasizing the importance of protecting users from phishing attacks. One of the main benefits of FIDO-based authentication is its resistance to those attacks. Because a website's domain is cryptographically bound to any FIDO credentials used on the site and the authentication process is challenge-response-based, attackers have no way of tricking people into using those credentials

[10] will appear at IEEE S&P 2023

on a fake site [99]. Users also cannot communicate their FIDO credentials via email, SMS, over the phone, or otherwise. Thus, even manual efforts like *spear phishing* or *voice phishing*[11] are pointless.

However, users might not know which authentication methods protect them from phishing attacks. For instance, all participants of a small pilot study investigating user awareness of phishing and WebAuthn thought traditional two-factor authentication (2FA) methods protected them against phishing [180]. Hence, educating and persuading users to replace their traditional authentication methods with a FIDO-based approach is vital.

Moreover, attackers will always target the weakest link to achieve their goal. If FIDO-protected accounts have phishable backup authentication options, attackers will entice users to use those instead of their phishing-resistant FIDO credentials. Research revealed that social engineering downgrade attacks against FIDO-based authentication are feasible with high success rates [183]. Ulqinaku et al. found that 55 % of study participants fell for a real-time phishing attack forcing an authentication method downgrade through social engineering, while another 35 % were found to be potentially susceptible to this kind of attack in practice [183]. For the study, a real-time phishing platform was built that downgraded FIDO-based second-factor authentication [183]. The authors also observed that all FIDO-supporting websites in Alexa's top 100 allow choosing alternate second factors, making them potentially vulnerable to similar downgrade attacks [183]. None of the study's participants indicated that they would rely on FIDO to detect phishing attempts. In fact, some participants had a false sense of security as they thought their accounts were protected because they had FIDO 2FA enabled, even though they were phished with one of the backup authentication methods during the user study [183]. No evidence was found that any of the 51 participants understood the underlying concept [183]. Ulqinaku et al. concluded that a refusal to accept alternative 2FA methods if the user has FIDO authentication enabled was the only way to effectively prevent downgrade attacks [183].

2.1.6. Multi-Device FIDO Credentials

Before 2022, all FIDO credentials – on both roaming and platform authenticators – were constrained to a single device and thus not transferrable. That is because for single-device FIDO credentials, the private key material is device-specific and stored in a secure element that prevents readout [85]. This was also the case for smartphones with iOS and Android operating systems. But, of course, like with most security measures, the benefit of added security is pitted against a UX drawback. For instance, users cannot synchronize their credentials between devices, risking losing access to accounts if a device is lost.

To improve UX and drive adoption, the *FIDO Alliance*, in coordination with W3C, proposed additions to the FIDO and WebAuthn specifications [69]. The proposal introduced multi-device FIDO credentials [69] that can be synchronized between devices within the same ecosystem. For example, users of Apple devices can synchronize their credentials from their phone to their tablet and desktop computer [14]. The same is true for other ecosystems like Google's Android operating system. Currently, synchronization is only possible within one of those ecosystems, though. The synchronization is possible because the private key material is no longer bound to the secure element of a single device [69, 85]. While this change results in a lower level of security, it also has the potential to eliminate one of the major usability issues: recoverability [85].

[11] impersonation over the phone

Along with the *FIDO Alliance*'s proposal went a joint public commitment by Apple, Google, and Microsoft to expand support for FIDO-based passwordless single-factor authentication. All three companies announced support for multi-device FIDO credentials in their operating systems [14, 27, 74, 98, 193]. Because "multi-device FIDO credential" does not exactly roll of the tongue easily, the companies refer to them with the consumer-friendly term *passkey*[12]. It is unfortunate that there are conflicting definitions of what a *passkey* is [69, 85], but this thesis will use the following distinction:

- A security key is a roaming authenticator that typically produces single-device FIDO credentials.
- A passkey is a multi-device FIDO credential typically produced by a platform authenticator.
- A single-device passkey is a single-device FIDO credential produced by a platform authenticator.

The latter is a special case that should rarely be relevant. For instance, older versions of iOS only support single-device FIDO credentials but not multi-device FIDO credentials. Newer iOS versions only support multi-device FIDO credentials, not single-device ones [97]. The removal of iOS's ability to generate single-device FIDO credentials that provide greater security has been criticized after Apple's presentation of passkeys [97]. The reasoning behind this decision is comprehensible, though. Support for both types may have confused users into thinking their credentials were backed up when, in fact, they were not. Security keys still are an option for those that appreciate the added level of protection of a single-device FIDO credential.

In addition to multi-device FIDO credentials, the *FIDO Alliance* also proposed a new protocol based on the Bluetooth standard to make smartphones capable of acting as roaming authenticators [69]. The protocol is a derivative of *caBLE*, an earlier proposal by Google [133]. The latest draft of the next CTAP specification version introduces the concept under a new *hybrid* authenticator transport method [31]. A client (e.g., a web browser) starts the authentication ceremony by displaying a QR code that the user has to scan with their smartphone's camera. The smartphone, acting as an authenticator, and the client device establish a BLE communication channel using a shared secret within the QR code's URI content. The use of Bluetooth is intended to guarantee physical proximity [31]. The actual transport of *CTAP2* messages happens over a high-availability tunnel network service with a domain name known to the authenticator [31]. This will typically be a service the phone's operating system (OS) vendor provides. The new *hybrid* transport makes it possible to use a passkey on devices outside their originating ecosystem. For example, a user could sign in on a Windows computer using Chrome when their passkey was stored on their iCloud keychain using their iPhone [69].

Since their announcement, passkeys have been widely reported on by various industry-specific news outlets, as well as large mainstream publications [38, 51, 71, 77, 108, 113]. Besides Apple, Google, and Microsoft, password manager vendors like AgileBits (1Password), or Dashlane have also announced support for passkey management [145, 190]. Aside from companies like Cloudflare or GitLab, which recently started exclusively requiring a FIDO credential as part of their employees' MFA process [75, 158], first companies like Shopify, Kayak, and Instacart have already integrated passkey support into their customer-facing applications [15, 120, 188].

[12] written in lowercase, like password

As Google rolled out passkey support to all Google accounts in the first quarter of 2023 [33], early data[13] suggests significant UX improvements compared to password-based authentication. Convento et al. suggest that passkey authentication has a significantly higher success rate of 63.8 % compared to 13.8 % for password-based authentication. While the average time it takes users to login with a password is 30.4 s, on average it only takes them 14.9 s using passkeys [52].

2.1.7. Conditional Mediation

As mentioned before, WebAuthn allows for two modes of operation: A FIDO credential can be used as a second factor in an MFA context, or as a single *passwordless* factor [32]. The former use case has seen comparatively wide adoption, while the latter is virtually never available in the real world – despite the unique feature of phishing-resistance [166]. Within the scope of the latest published version of the WebAuthn specification, it is impossible for relying parties to detect whether a FIDO credential is available on a client [32]. This is by design, because disclosing this information to relying parties would impair user privacy [166]. However, relying parties do not want to worsen UX by performing a WebAuthn request if there is a good chance for a visible error to appear in the client's user interface (UI) since no credentials were available [166]. Websites may add a button to their website to let users manually trigger a WebAuthn API call, which would burden and potentially confuse users. Additionally, Satragno and Hodges found that implementers were unsure how to label said button, since users *may* know their platform-specific authenticator's name (e.g., *Windows Hello* on Windows or *Touch ID* on iOS) but not the technical, cross-platform terms WebAuthn or *FIDO2* [166].

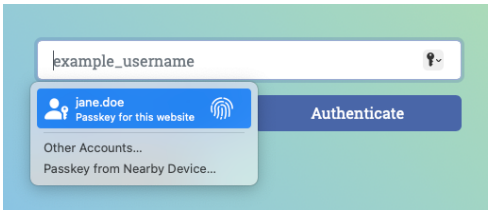


Figure 1: WebAuthn Conditional UI

The latest editor's draft of the WebAuthn specification introduces *conditional mediation* to fix this UX issue [101]. Along with the technical concept, user agents like Chromium and Safari added support for the user-facing *Conditional UI* [106, 166]. An example in Safari on macOS is shown in fig. 1. *Conditional mediation* allows users to select from a list of discovered credentials when focusing on an input field within a sign-in form without the website being aware of the credentials' existence until the user chooses to authenticate with them. This behavior is similar to a password manager's auto-fill functionality.

Listing 4 shows a minimal WebAuthn call using *conditional mediation*. Implementers can use a new static `isConditionalMediationAvailable()` method (lines 1-2) to test whether a user agent supports *conditional mediation* [101]. If it does, a `navigator.credentials.get()` call can be made using the mediation configuration value `conditional` (line 8) [101]. The resulting promise only returns if the user actively selects a credential discovered by the client and possibly unlocks it using a PIN or biometrics. When no credential is found, or the user dismisses the *conditional UI*, the promise is never resolved [101, 166]. Timeout values should hence be ignored [166]. Thus, the API call can be invoked when the login page has loaded because it does not fail

[13] March - April 2023, $N \approx 100M$

visibly while user privacy is preserved with silent errors [166]. In certain scenarios, for instance, when reauthenticating a known user to perform a risk-bearing action (e.g., changing account preferences), an RP may set the `allowedCredentials` property to filter discovered credentials to the ones associated with the currently signed-in account [166].

Listing 4: Conditional UI JavaScript Example (Adapted from [166])

```
if (!PublicKeyCredential.isConditionalMediationAvailable ||
  !PublicKeyCredential.isConditionalMediationAvailable()) {
  // Conditional UI is not supported by the browser.
  return;
}

navigator.credentials.get({
  mediation: 'conditional',
  publicKey: {
    challenge: randomStringFromServer,
    // `allowCredentials` can be used to filter results,
    // e.g. if user has to reauthenticate.
  }
});
```

To allow the user agent to show the *conditional UI* on the correct input fields, implementers can add the HyperText Markup Language (HTML) autofill `webauthn` token, for example, to existing username and password input fields. An example is shown in listing 5. Using the autofill token, RPs can gradually roll out passkey support to their users without cluttering their authentication UI and possibly confuse users. In regard to UX, users can be introduced to the use of passkeys through a familiar interface [166].

Listing 5: Conditional UI HTML Example (Adapted from [166])

```
<label for="name">Username:</label>
<input type="text" id="name" name="name"
  autocomplete="username webauthn">

<!-- Could be ommitted eventually if only passwordless authentication was supported. -->
<label for="password">Password:</label>
<input type="password" id="password" name="password"
  autocomplete="current-password webauthn">
```

2.2. Distributed Web Crawlers

Apart from available and commonly used means of authentication, the second theoretical foundation for this thesis is research on crawling the WWW. This section presents various web crawler architectures described in the literature and their shared and distinctive concepts.

2.2.1. Use Cases

Web crawlers have a wide range of applications in various fields. One of the most notable uses is in search engines, which collect and organize (HTML) documents to enable users to query for specific content [37, 142]. Similarly, crawling can be used for archiving purposes where parts of the web are regularly downloaded and archived to ensure the permanent existence and availability of information [142]. The best-known example is the non-profit organization *Internet*

Archive, which open-sourced its custom crawling engine *Heritrix* [94, 118]. Another use case is the uptime monitoring of a web service. Businesses may want to ensure the constant availability of their website from different continents or get notified if a Transport Layer Security (TLS) certificate's expiration is pending. Lastly, an important application of web crawlers is data mining [142]. The extraction of data from a website may be used for a variety of applications. For example, services may provide notifications for users when the price of a product available at an online shop drops below a configured threshold. Texts from online encyclopedias like Wikipedia are also used to train *natural language processing* models. In addition, data can be used for statistical purposes, which is also the goal of this thesis.

2.2.2. Fundamentals

Before discussing various web crawler architectures described in the literature, the basic considerations behind any crawler need to be introduced.

Data Structures

A web crawler typically has two data structures that determine its state. Firstly, it keeps track of a set of discovered Uniform Resource Locators (URLs) and whether they have been visited or not. Olston and Najork call this data structure the "URL-seen test" or the "duplicated URL eliminator" [142]. Secondly, a set of URLs that have not yet been visited. Olston and Najork call it the *frontier* [142]. The former data structure must support set addition and set membership testing [142], while the latter must support adding URLs and selecting the next URL to crawl [142]. For example, the *URL-seen test* can be implemented with a hash table or a bloom filter [29, 142].

The simplest implementation of a *frontier* data structure is a First-In-First-Out (FIFO) queue [142]. However, since a significant share of hyperlinks is relative, a FIFO queue will typically contain many consecutive URL entries containing identical domains when crawling in a *breadth-first* fashion, i.e., visiting all discovered subpages before continuing with the following website [142, 146]. Sending lots of requests in a short amount of time, or even in parallel, is considered "impolite" and may be equivalent to a *(distributed) denial-of-service attack* [142]. To prevent such behavior, web crawlers should impose politeness policies detailed in section 2.2.3.

Apart from the crawl state, the collected content is stored in a *repository* for subsequent examination [142]. For instance, the content may be stored in a key-value fashion, identified by a key derived from its URL [18].

Essential Components

For a general-purpose web crawler to operate, it needs several components. Besides the aforementioned URL *frontier* that stores a list of download targets [142], a Domain Name System (DNS) resolver is indispensable to retrieve the responsible web server's Internet Protocol (IP) address [87]. Another component is needed to download content using HTTP. Next, the obtained HTML document must be parsed, and any hyperlinks need to be extracted [87]. Finally, the *URL-seen test* component needs to eliminate known URLs to prevent repeated visits and possible loops [87, 142].

Crawl Ordering Problem

Because of its continuing growth and the use of dynamic[14] content, the amount of crawlable web pages can practically be considered infinite [17]. For example, for a search engine to prioritize downloading URLs with high page rank, it needs to establish a crawling order [142]. When defining a crawling order, two diverging goals must be weighed against each other: *Coverage*, meaning the represented fraction of available pages, and *freshness*, i.e., how up-to-date the collected content is kept [142]. To allow for repeated downloads of content, Olston and Najork distinguish two approaches:

- **Batched Crawling.** During a crawling process, URLs are not crawled repeatedly. Instead, a new iteration of the process is started to obtain more recent snapshots of all desired pages [142].
- **Incremental Crawling.** Crawling is a continuous process that never stops. URLs may appear multiple times in the crawl order to refresh their content [142]. Incremental crawling is considered more powerful because it allows for page re-visitation at different rates, which is useful for a wide range of applications [45, 142].

Scoped Crawling

While *comprehensive crawlers* like general-purpose search engines may want to capture any (popular) available content on the web, *scoped crawlers* only consider a specific subset of pages or content relevant to their use case, which allows crawling to be significantly faster and cheaper [142]. The *scope* of a crawler could be based on a specific language (e.g., pages in German), a geographic region (e.g., content published in Europe), a content type (e.g., images and videos), a page type (e.g., online shops), a topic (e.g., pages about gardening), or any combination of other aspects [142]. Especially for data mining applications, scoped crawling can be advantageous [142]. For example, an aforementioned price comparison crawler would try only to visit online shops to extract available products and their prices.

Avoiding Troublesome Content

When building a web crawler, one needs to expect encountering content that is undesirable or problematic [142]. While trying to prevent crawling duplicate content resulting from URL aliases that refer to the same content or mirrored content [87] is apparent, there are other forms of content that should be avoided. For instance, *crawler traps* inflate the web corpus without information gain [87, 142]. Some crawler traps are non-malicious, for example, a web-based appointment booking tool that uses a unique path for each month and dynamically generates hyperlinks for the following month. This would constitute an unbound chain of dynamically generated pages [142]. Other traps are malicious, like spammers that want to influence a search engine's ranking of their website by creating spam content linking to their website [142]. Another problem is *cloaking*, i.e., serving different content to web crawlers and human visitors [142]. Cloaked content is not necessarily served for malicious motives. For instance, many websites rely heavily on the use of JS, which many web crawlers cannot interpret [142]. Thus, serving static content without relying on JS can benefit website and crawler operators alike. However, nowadays, a significant number of websites employ some form of bot detection to prevent things like credential stuffing attacks, but also content scraping [47]. In fact, one of the largest vendors for bot detection is content delivery network (CDN) provider *Cloudflare*, whose CDN was found

[14] in this context meaning generated, not client-side rendered

to be used by 52 % of sites employing a CDN (29 %) to serve their HTML content in the *HTTP Archive*'s 2022 *Web Almanac* report [25]. There are many techniques used for bot detection – some simple, like comparing a client's *User Agent* [142], some advanced, like behavioral analysis based on *Machine Learning* [47]. Chellapilla and Maykov studied *redirection spam*, which is a form of content cloaking where false content is served on a website along with an immediate or delayed JS-based redirect [43]. They found that around half of sites employing redirection spam obfuscate the redirection target URL, making a static analysis challenging [43]. Instead, they proposed the use of a light-weight JS interpreter [43]. In 2010, Olston and Najork identified dealing with client-side rendered content as one of the main upcoming challenges in web crawling [142]. Although there are viable ways, crawling this content remains challenging and is more resource-intensive. Section 2.2.5 discusses the means and hurdles of client-side rendered content crawling further.

2.2.3. Politeness Policies

Because web crawlers should be good "Internet citizens", they should behave in a polite manner towards service providers and website operators [142]. To prevent overwhelming web servers, a *politeness policy* should be defined [142]. A recent example of the failure to enact a politeness policy was an incident where the *Internet Archive* repeatedly became unavailable for about an hour at a time, after an unnamed Amazon Web Services (AWS) customer launched tens of thousands of requests per second from 64 hosts on the virtual compute platform [102]. This exhaustive behavior must be avoided, as it causes massive problems for service providers and potentially prolonged unavailability for all other users.

The need for politeness was recognized early on after the WWW became publicly available. In 1994, the initial version of the *Robots Exclusion Protocol* was defined [111]. Today, RFC 9309 governs how crawlers are supposed to respect the wishes of website operators. It specifies simple, grouped rules to be served in a text file reachable at the domain's root path `/robots.txt` [111]. Listing 6 shows an exemplary rule set where all crawlers are asked not to download the URIs `/login` and `/api/`. Specifically, a crawler with the user agent `BadBot` is asked not to visit any page. However, the *Robots Exclusion Protocol* depends entirely on all parties honoring it. The rules do not provide any form of access restriction [111].

Listing 6: Robots Exclusion Protocol Example

```
1 user-agent: *
2 disallow: /login
3 disallow: /api/
4
5 user-agent: BadBot
6 disallow: /
```

When defining a politeness policy, crawling implementers want to avoid overwhelming web servers while also not being wasteful with their resources. For instance, *WebCrawler*, which was one of the early web crawler implementations in 2000, used a *breadth-first* algorithm that has the desirable side-effect of automatically creating delays between subsequent visits to a single website, which was appreciated by server administrators [146]. However, crawlers should not waste their computing and network resources when delaying requests in any way. Instead, other meaningful work should be done for the duration of the delay [142]. For example, instances may run numerous breadth-first crawling processes in parallel.

The easiest politeness policy is to not issue multiple overlapping requests to the same web server [142]. Since, for almost any large-scale crawling task, work needs to be distributed to multiple instances, partitioning said work by the host component of URLs makes a non-overlapping politeness policy easy because no communication between instances is necessary [142].

A more conservative, more complex approach would be to space out requests based on the web server's capacity [142]. For example, Heydon and Najork's *Mercator* crawler utilized a delay of subsequent requests by a multiple of the previous server response time. The multiplication factor was called the *politeness parameter* and was configurable (e.g. 10× response time) [87].

2.2.4. Building for Scale

With an ever-growing WWW, developing a web-scale crawler presents significant engineering challenges, all of which revolve around the aspect of scalability [136]. For example, if a search engine were to index ten billion web pages and keep their content reasonably fresh with an average update window of 4 weeks, its crawler would need to download over 4.000 pages per second [136]. To achieve these kinds of download rates, crawlers need to be distributed over multiple machines, with each one performing multiple downloads in parallel [136].

Some web crawler designs like Brin and Page's early architecture for the Google crawler in 1998 distribute the work of downloading among multiple instances, while the sets of discovered and downloaded URLs are maintained on a single machine [37, 136]. The centralization of these major data structures is appealing because of its simplicity, but it ultimately becomes a bottleneck, thus the scaling potential is limited [136].

Another early web crawler was Pinkerton's *WebCrawler* that was initially introduced in 1994. *WebCrawler* was organized into a central crawl manager and 15 crawling instances [146]. The initial architecture did not prioritize scalability, as it used a general-purpose database management system (DBMS) to store crawling state [146].

In 1999, Heydon and Najork presented the first version of *Mercator*, an incremental crawler design blueprint that initially scaled vertically and was non-distributed [87]. Back then, machine performance also was a significant bottleneck, as retrieved data was too big to fit in memory entirely, which required balancing the use of hard disk and memory space [87]. *Mercator* was able to download 46.3 HTML documents per second with an average page being 5 kB [87].

Edwards et al. introduced the distributed, incremental *WebFountain* crawler in 2001 [62]. It was characterized by the absence of a global scheduler, global queues, or the ability for one machine to access a global list of URLs [62]. It had three major components: machines dedicated to downloading content called *Ants*, *duplicate detectors*, which rejected (near-)duplicates, and a single control-plane machine named the *Controller*, which was responsible for tasks like routing discovered URLs, load balancing, and monitoring tasks [62].

When Najork and Heydon presented the second, now distributed, version of *Mercator* in 2002, it partitioned the URLs to crawl using their hostname component. Thus, the potential bottleneck of a centralized set of URLs was avoided [137]. While experimenting, Najork and Heydon found that their so-called *weak politeness guarantee*, where only one thread was allowed to contact a particular web server at a time, was still considered too "rude". Server administrators who issued complaints were troubled that no pauses were made between subsequent requests [137]. Consequentially, Najork and Heydon built a more sophisticated *URL frontier* implementation that ensured pauses between subsequent same-page requests and additionally allowed for URL prioritization [137]. The paper argues that *checkpointing*, i.e., persisting task progress state to

allow for recovery in the event of an error, is paramountp for operating a long-running process like a web crawl [137]. In their paper, the authors also presented statistics on a 17-day long crawl that was performed in the year 2000, which had processed 891 million URLs [137]. Because of its extensibility, *Mercator* was later used as a blueprint for various data mining projects [142].

In 2004, Loo et al. presented a fully-distributed peer-to-peer crawler architecture where nodes can operate independently and are coordinated by a Distributed Hash Table (DHT) [119]. DHTs are particularly useful for this application because of their automatic load balancing and reliable content-based routing [119]. Each crawling instance is responsible for the URLs published in its partition of the DHT [119]. The authors also studied different strategies for partitioning crawling tasks and found that partitioning by hostnames and falling back to URLs as partition keys if a crawler's pending input queue size exceeded 500 tasks allowed for the highest throughput [119].

When Boldi et al. showcased the fully-distributed, fault-tolerant *UbiCrawler* that same year, they introduced *consistent hashing* [103] to uniformly distribute work between crawling agents. Each URL's host component is used as key, while crawling agents are considered hash buckets. A point on the *unit circle* is computed from the key to determine which agent is responsible. The nearest bucket on the circle dictates which agent will crawl the URL [30]. A linear relationship between the number of agents and the number of crawlable pages was found as a result of this distributed coordination logic [30].

Bahrami et al.'s 2015 proposal is the first cloud-based architecture found in the literature on scalable web crawlers [18]. The authors' design uses cloud computing features and the *MapReduce* programming model [18, 55]. Their implementation uses the vendor-specific products offered by cloud provider *Azure* [18]. The architecture is based on *Azure Cloud Queue* to maintain a temporary list of URLs to crawl and *Azure Cloud Table* to persist information on crawled URLs [18]. The architecture design is not reliant on a central coordinator. Instead, a crawling agent can start more agent instances on-demand [18]. When the crawl process starts, the Cloud-based Web Crawler Engine (CWCE) boots the first agent instance, which creates and fetches the first URL from the *DNS Resolver*. If the URL is not in the queue and has not been visited yet, it is added to the queue and the table as an unvisited URL [18]. *Azure Cloud Table* is a wide-column store, i.e., a column-oriented, NoSQL-based DBMS comparable to *Google*'s *Bigtable* [42]. Distributed wide-column stores like *Azure Cloud Table* partition content between machines using a partition key and index content using a row key. Together, they act as a primary key [18]. Bahrami et al. use the URL's host component as a partition key and a hash value of the URL as the row key. That way, any content originating from one hostname is stored on the same database instance, which makes queries faster [18]. Besides *Azure Cloud Table* and *Azure Cloud Queue*, Bahrami et al. make use of *Azure Blob Storage* as an archive for large, unstructured data like PDF files, videos, or images [18].

In the same year, Quoc et al. introduced *UniCrawl*, a geographically distributed crawler that also makes use of the *MapReduce* paradigm [55] and is based on the architecture of *Apache Nutch* [135, 149]. The workload, i.e., the "domain space", is distributed over several geographically distributed sites, which the authors claim could reduce capital and operating expenses, for instance, by allowing multiple small companies to share a common crawling infrastructure [149]. For storage, *UniCrawl* utilizes *Infinispan* [175], a distributed key-value store that makes use of *consistent hashing* [103] and supports features like a one-hop routing design, built-in replication, and elasticity that allows more nodes to join the ring structure [149]. In comparison with a baseline technique where a central crawler is simply stretched over multiple locations, the authors found *UniCrawl* to have a performance improvement of 93.6 % in terms of network bandwidth consumption and a speedup factor of 1.75 [149].

Prusty et al. built a horizontally scalable crawler using Docker containers and Kubernetes orchestration, allowing users who cannot code to commission scoped web crawls in a web-based UI [147]. The architecture consists of a main application serving the UI and an API, which stores data in a central MySQL database. For every job request, the main application creates separate *crawler manager* container instances along with a local Redis cache container. *Crawler managers* receive the list of URLs to crawl from the main application. The *crawler manager* instances then spawn a user-configurable amount of dedicated crawler instances and supply them with target URLs. The crawler instances use a global Redis cache to maintain a list of recently crawled URLs to avoid unnecessary repeated visits. After crawling, the fetched responses are stored in a cloud storage bucket compatible with the *S3* API [9, 147]. The authors argued that using a separate *crawler manager* instance avoided complexity problems and allowed for better horizontal scalability [147]. Prusty et al. also compared means of crawling dynamic, client-side rendered content, which are discussed in section 2.2.5.

2.2.5. Crawling the Dynamic Web

In 2010, Olston and Najork identified the question of how to crawl client-side generated, dynamic content as a future direction for web crawling research that had received almost no attention, except for one piece of preliminary work by Duda et al. [142]. Indeed, increasing adoption of dynamic content generated by client-side JS has been an unbroken trend ever since. While 49.1 % of surveyed websites did not use any JS in 2012, this share decreased to only 17.9 % in 2023 [148]. Dynamic, client-side rendered content makes it hard for traditional, static crawlers to extract relevant information from pages [60, 142, 147].

When Chellapilla and Maykov studied *redirection spam* in 2007, they advocated the use of a lightweight JS parser along with a tuned execution environment for predicting whether redirection behavior would occur. However, this concept of JS parser and execution environment would only be intended to fulfill the exact use case of detecting redirection [43].

In 2008, the previously cited work by Duda et al. presented *AJAXSearch*, a prototypical search engine for dynamic web content [60]. It used a breadth-first approach that triggered all available events on a crawled page and invoked corresponding JS functions in an execution environment. Whenever a Document Object Model (DOM) change occurred, a new state was created, forming a (deduplicated) state machine graph [59, 60]. In comparison with a traditional crawling approach, using *AJAXCrawler* with cached responses was $\approx 10\times$ slower when tested against the *YouTube* website. The crawling time for 10,000 pages with *AJAXCrawler* was around 68 h [59]. Unfortunately, no further development or usage of *AJAXSearch* can be found in the literature.

As previously described, Prusty et al. compared several ways of crawling dynamic web content relevant to their Python-based technical stack [147]. The first evaluated method was *PyQT* [157], a Python wrapper around the *QT framework* that acts as a browser and can be used to render dynamic web content without the need for a full external browser engine. However, the authors decided not to use *PyQT* because of a lack of available documentation and the inherent complexity of the *QT framework* [147]. Another evaluated option was *Splash* [110], a lightweight web browser with an HTTP API that is partly based on and abstracts the *QT framework*. It is considered very fast at rendering JS pages and can be run in a Docker container alongside one's application. Prusty et al. ultimately decided against using it, even though it had slightly better performance results than their chosen method. They argued that *Splash* would have required adding another central containerized service to their architecture or having a *Splash* container inside every crawler instance container. As both would have increased complexity and

added other undesired side effects, they ultimately chose *Selenium* instead [147]. Selenium [173] is a versatile programming library with extensive community support that remotely controls a complete web browser. Compared to *Splash*, it does not require running in a separate service. The authors also note that it did not increase communication overhead within their architecture, which they were trying to avoid [147]. In the end, they chose to use *Selenium* along with the *Chrome* driver. Since they deployed a containerized crawler, deployment was as easy as installing the *Chrome* package in the Docker image [147].

Apart from the described studies, though, dynamic web content crawling is still underrepresented in the available literature, considering that 82.1 % of websites use some form of JS library [148].

3. Architecture

This section addresses the high-level software architecture of the distributed web crawler presented in this work. First, some demand estimations for compute and storage resources are calculated based on the defined scope and requirements. Subsequently, the distributed system components and the crawl process design are reviewed.

3.1. Scope and Requirements

To determine whether an authentication technology like WebAuthn is used on a website (R1), the website's content needs to be crawled and analyzed. Specifically, all HTML and JS resources that are included on any page that authenticates users can be considered relevant for analysis. This means that it is not necessary to crawl, store and analyze every page of a targeted website. Thus, a *scoped crawler*, as described in section 2.2.2, is appropriate for the intended application. In the context of this thesis, a singular content corpus is needed for subsequent analysis. Therefore, it makes sense to deploy a batch crawler[15].

Section 2.2.5 mentions that a significant share of websites, 82.1 % [148], use some form of client-side rendered, dynamic content and that not being able to capture it could skew results. Several approaches are conceivable to study the detection rate difference between a static web crawler and a dynamic crawler capturing client-side rendered content (R2). A lightweight JS parser and execution engine, as proposed by Chellapilla and Maykov [43], could improve content capture quality while using compute, memory, and storage resources as sparingly as possible. However, to get the most comprehensive picture, it is crucial to let a crawler behave as closely to a real web browser as possible, similar to Prusty et al.'s approach [147].

As previously noted, the crawler can be scoped to only fetching HTML and JS resources on all pages where users may authenticate on a website. Hence, a detection mechanism for whether a linked page could authenticate users must precede subsequent visits when extracting links from a domain's home page. Hence, a detection mechanism as to whether a linked URL could serve the purpose of authenticating users must precede subsequent visits when extracting links from a domain's home page. While the actual fetching of content requires differing implementations for statically and dynamically crawling websites, some features, including authentication URL detection or subsequent content analysis, may share the same software components.

To enable crawling a finite number of websites, a dataset with a limited number of domains must be used, which at the same time is representative of the web in general. Section 4.1 details the available data sets and their respective benefits and drawbacks. However, the size of a target data set also impacts potential crawler architectures in terms of scalability requirements.

Since many websites are to be surveyed, the crawler needs to fetch content concurrently to complete a crawl in a reasonable time frame. Section 3.1.1 gives some rough estimations on a duration comparison. While many requests need to be sent concurrently, the crawler must ensure it adheres to a politeness policy, as described in section 2.2.3, not to overload web servers or be perceived as a disturbance.

[15] as opposed to an incremental crawler, see section 2.2.2

3.1.1. Napkin Math

A rough estimation of some basic metrics helps define the technical requirements' order of magnitude. While outliers in either direction are virtually guaranteed, averaged content size and request duration measurements from previous research provide an initial foundation for further analysis.

Website Size

According to *HTTPArchive*'s 2022 *Web Almanac* report, the median total web page size on a desktop is 2.3 MB [93]. This includes any loaded HTML, JS, styling, and images. Due to the defined scope, only HTML and JS responses with respective median response sizes of 31 kB and 509 kB [93] need to be persisted. This results in a median content weight w of 540 kB per page for desktop.

Since there is no good way to estimate this without analyzing existing data, let an initial guess for the number of URLs p_g that need to be crawled per website be three. This includes the home page itself and any authentication-related URLs the home page links to. Depending on the actual URL detection methods, some websites may not have matched links, while others may have dozens. Nevertheless, this guess can help to establish a rough initial order of magnitude.

Also, let the targeted domain set T have a size of 1,000,000. To investigate R2, every target domain needs to be crawled by a static and a dynamic crawler, so $C = \{\text{static}, \text{dynamic}\}$. Because the expected response data is text-based and shares some common patterns like typical HTML boilerplate code, a conservative compression ratio c_r of 1.5 is estimated. If every response is stored and no de-duplication is applied, eq. (3) shows that the total corpus size s_c would be 2.16 TB. Consequently, this is the minimum available storage space a deployment environment should have.

$$s_c = \frac{(w_{\text{HTML}} + w_{\text{JS}}) \times p_g \times |T| \times |C|}{c_r} = \frac{(509\,\text{kB} + 31\,\text{kB}) \times 3 \times 10^6 \times 2}{1.5} = 2.16\,\text{TB} \quad (3)$$

Number of Requests and Crawl Duration

The 2022 *Web Almanac* also found that the median desktop page load consists of 76 individual HTTP requests [93]. Breaking those requests down by response content type, the crawler's scope leads to the 22 requests for JS and 3 requests for HTML per page being relevant (r_{HTML} and r_{JS}) [93]. When assuming that a content type can be inferred before a request has been made, the remaining requests can be skipped to improve crawler performance.

The set of targeted domains T is assumed to be comprised of the most-visited websites, which presumably value UX and thus serve content with a reasonable Time to First Byte (TTFB), i.e., the time passing between sending an HTTP request and receiving the first byte of the response. *Google*'s *Chrome* developer team suggests most sites should strive for TTFB durations of 800 ms or less [187]. As a rough estimate, let every request's latency l be 800 ms. Then, eq. (4) shows that the crawler would have to perform a total number of 150 million requests r_t, assuming three pages would have to be visited on every website (p_g) and every page would be fetched by a static and a dynamic crawler (C). If every request takes 800 ms, sequential processing t_s would take over 3.8 years. Of course, it makes sense to distribute the work across multiple machines where every machine makes several requests in parallel [142]. If $i = 100$ instances crawled

simultaneously, the process would take around $t_p \approx 14$ days to complete. If 200 instances were used, the crawl would only take approx. days.

$$
\begin{aligned}
r_t &= (r_{\text{HTML}} + r_{\text{JS}}) \times p_g \times |C| \times |T| = (3 + 22) \times 3 \times 2 \times 10^6 = 150 \times 10^6 \text{ requests} \\
t_s &= r_t \times l_s = 75 \times 10^6 \times 800\,\text{ms} = 6 \times 10^{10}\,\text{ms} \approx 1{,}388.9\,\text{d} \\
t_p &= \frac{t_s}{i} \\
t_{p100} = \frac{6 \times 10^{10}\,\text{ms}}{100} &= 6 \times 10^8\,\text{ms} \approx 13.9\,\text{d} \quad \text{and} \quad t_{p200} = \frac{6 \times 10^{10}\,\text{ms}}{200} = 3 \times 10^8\,\text{ms} \approx 6.9\,\text{d}
\end{aligned}
\tag{4}
$$

3.2. Selecting System Components

The chosen crawler architecture shown in fig. 2 is inspired by Bahrami et al. [18]. Especially the design of data structures and the content partitioning approach discussed in sections 3.3.2 and 3.3.3 are based on their proposal. One goal was to employ components that scale well horizontally and are commonly used in cloud environments. Figure 2 shows *workers*, which are crawling instances handing tasks at a sub-process level. Since they do not depend on a singular coordination instance, the number of workers can be easily scaled horizontally by increasing the number of workers per machine or the number of machines running in the cluster. Workers use identical binaries and are configurable to crawl statically or dynamically with a remote-controlled Chrome process. To receive tasks, workers listen on a *task queue*. Crawled content is stored in a *wide-column store*. A single *seeder* instance is responsible for filling the task queue with targets from a pre-compiled list of domains. To ensure uptime and proper performance, a monitoring stack (consisting of Prometheus and Grafana) monitors all relevant instances.

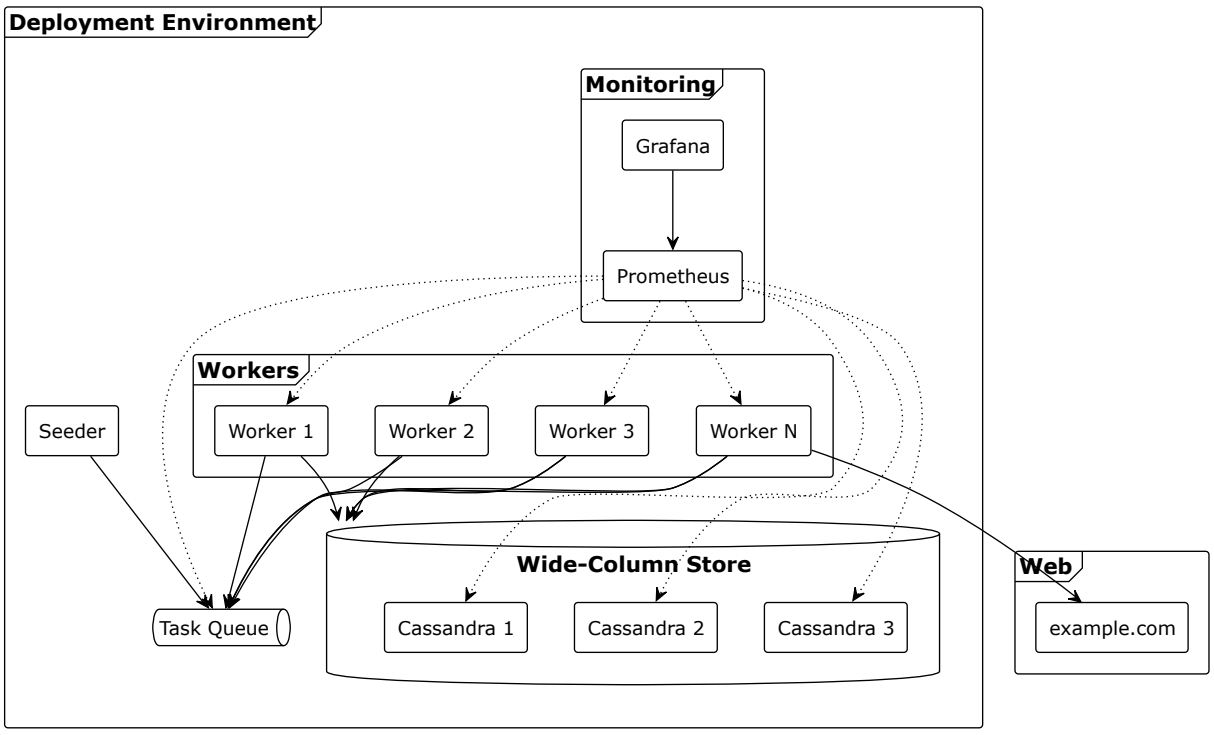


Figure 2: High-Level Architecture Overview

3.2.1. Queueing

Najork and Heydon correctly identify *recoverability* (or *checkpointing*) as an essential part of any long-running process like web crawling [137]. In the event of an error, a web crawler must be able to recover and continue its work without losing large amounts of its achieved progress. Since web crawling can be a complex endeavor, errors are likely to occur at some point. The described architecture utilizes queue task acknowledgments to prevent tasks from getting lost through worker failures. A queue component thus must somehow support acknowledgments to work within the overall architecture. From the various available open-source queueing software options, Redis and RabbitMQ were selected for closer evaluation because of their ease of use, ease of deployment, widespread community support, and quality of available documentation.

While Redis is a simple in-memory data store, it has many use cases. For example, it may be used as a database, cache, or message broker [162]. The benefits of Redis include a wide range of available client libraries in various programming languages and excellent performance, as data structures are stored in memory. Redis can also persist data to permanent storage to survive reboots and failures. Additionally, it can be scaled horizontally based on hash-based sharding[16] [162]. Although some third-party client libraries support task acknowledgments, Redis itself does not. Unfortunately, relying on third-party support for this architecturally essential feature would result in dependence on younger, less widespread libraries that are not as seasoned and well-known as Redis itself.

RabbitMQ, on the other hand, is a dedicated message broker that is widely adopted in the industry and is well-documented. It features flexible message routing, horizontal scalability, guaranteed message persistence, a wide range of client libraries, and delivery acknowledgement [107]. Although using RabbitMQ's core protocol *AMQP 0-9-1* is more complex, the plethora of available features and options make using it worthwhile. Particularly because of first-party acknowledgment support and rerouting options for negatively acknowledged messages, RabbitMQ was ultimately chosen for queueing crawling tasks in this architecture.

3.2.2. Data Storage

Since the permanent data storage of the crawler should not become the bottleneck of the application, a requirement for a storage component was to be horizontally scalable. Thus, more than one instance could ingest crawled content sent by the workers. The component should also be able to run in any cloud environment, so Microsoft's proprietary *Azure Cloud Table* used by Bahrami et al. was not a great fit, especially because the final deployment target was not decided on by the time the crawler's architecture was designed. Apart from scalability, low write and read latencies were also desirable. Traditional relational database systems based on Structured Query Language (SQL) are mostly not designed to be scaled horizontally. They also are not designed to store blobs of several MB in size. Due to its distributability and performance, an open-source wide-column store similar to *Azure Cloud Table* that can run in any environment looked the most promising: Apache Cassandra. Cassandra uses consistent hashing to distribute data uniformly across a dynamically scalable number of database instances. There is no main coordination node in Cassandra, either. Instead, every node in the cluster is identical, which prevents single points of failure and network bottlenecks [12].

Furthermore, because the Cassandra documentation states that a blob may be up to 2 GB in size[17] [53], and Bahrami et al. utilized the *S3*-compatible bucket storage *Azure Blob Storage*

[16] not *consistent hashing*
[17] less than 1 MB is recommended, though

only for large files, such as video, binary files, and images, the decision was made to refrain from using a bucket storage component in order to reduce complexity and save all crawled HTML content as a blob in Cassandra. Section 7.2.5 discusses why this decision caused problems later on and is not advisable.

3.2.3. Crawling

To investigate R2, the crawling process was divided into two separate approaches: traditional static HTTP crawling and dynamic crawling to capture client-side rendered content.

Implementing static crawling is relatively straightforward. With the right programming language (see section 4.2.1), the built-in HTTP client can be used to directly issue HTTP requests to the targeted website. Subsequently, one has to parse the responding HTML document to extract directly embedded or referenced JS sources.

Dynamic crawling is more complex, as the website's JS has to be evaluated in some execution environment. As section 3.1 lays out, mirroring the behavior of a real web browser comes closest to how a user would interact with websites. While there are ways to partly emulate a web browser when executing fetched JS, the best way is to utilize a full-featured web browser. Fortunately, Chromium-based browsers support remote controlling through the *Chrome DevTools Protocol* [78]. The protocol allows third-party software to instrument, inspect, debug, and profile supporting browsers. Oversimplified, one can make the browser do anything a user could without user interaction. Several popular libraries like *Puppeteer* [122] use the *DevTools Protocol* to remotely control a Chromium-based browser. In turn, these libraries may be used to automatically capture screenshots, generate PDF documents, automatically perform end-to-end user tests in a software development toolchain, or crawl Single-Page Applications (SPAs).

As all incoming and outgoing network connections can be monitored and intercepted using the *DevTools Protocol*, the dynamic crawler does not need to manually extract JS URLs referenced in HTML documents. Instead, it lets a Chromium-based browser visit the website and captures all incoming responses, as well as the page's rendered DOM.

The architecture anticipates the same crawler binary to be used for both static and dynamic crawler instances. Thus, when deploying on machines that are supposed to crawl client-side rendered content, a Chromium-based browser has to be installed alongside the binary so that the *DevTools Protocol* can be used to start and remotely control a web browser instance on the same machine.

3.3. Process Design

As section 3.1 states, the scope of the crawler is confined to a domain's home page and subsequent pages where users are authenticated. Letting a single *worker*, i.e., thread listening on a task queue, handle a complete target domain – meaning a website's home page and all subsequent pages it finds links to on the home page – is a simple approach that reduces complexity. It also allows the implementation of a straightforward politeness policy: If only one *worker* is responsible to crawl a website at a time, no coordination between *workers* is necessary to ensure proper rate limiting. This reduces complexity further, as no distributed locking mechanism is needed. With proper concurrency management, *workers* may even add a simple waiting period between subsequent requests to reduce stress on targeted web servers. Section 4.2.1 discusses the concurrency management differences in suitable software stacks.

3.3.1. Defining the Sequence of Operations

To satisfy R2, each target must be crawled independently using two different methods. However, using multiple crawling methods for the same target cannot negatively impact the straightforward politeness policy described earlier. The easiest solution is to create two task queues and let the static crawler dispatch a dynamic crawling task once static crawling of the targeted domain is completed. The high-level order of operations for statically crawling a website is defined as follows:

1. Get the next target domain from the task queue.
2. Fetch the `robots.txt` and extract possible authentication URLs that website operators might have specified to be excluded from search engine indexing.
3. Try to visit the home page.
4. If no successful TLS connection can be established or some other error occurs, store the error and start to work on the next task.
5. Otherwise, parse the HTML response and extract all anchor tags, e.g., `<a href="/login">Login</a>`, and script source URLs, e.g., `<script src="/script.js">`.
6. Store the HTML response of the home page.
7. Match anchor tags based on their label or the URL in their `href` attribute against a list of possible authentication page link texts and URIs. For reference, the regular expression patterns for authentication URLs are displayed in listing 13 in the appendix.
8. Deduplicate the URLs of all matched possible authentication URLs.
9. For each identified page, try to visit the URL.
   a) Parse the HTML response and extract all script source URLs.
   b) Store the HTML response.
10. Deduplicate the list of script source URLs.
11. For each script source URL, fetch and store the content.
12. Enqueue a dynamic crawling task for the targeted domain.

For the most part, the dynamic crawler performs these operations the same way. However, there are some important differences. For instance, item 5 only partly applies to the dynamic crawler because it skips extracting script source URLs. This step is unnecessary because it can intercept all requested script responses in a remote-controlled web browser. Section 3.2.3 details how interception works. Similarly, items 9a, 10 and 11 differ in that the dynamic crawler does not need to perform the extra step of extracting and fetching script source URLs. It will also skip fetching the `robots.txt` in item 2 and dispatching subsequent tasks like in item 12, as content analysis is manually triggered later on.

3.3.2. Designing Data Structures

There are two main data stores in the crawler's architecture. Two FIFO queues exist for crawling websites statically and dynamically. Each crawler type listens to its respective queue to obtain new tasks. The data structure of a task is very simple, it only contains the targeted domain. As section 3.2 describes, the *seeder* fills the static crawling queue with tasks. Each task is processed

by a static crawler as outlined in section 3.3.1. Then, the static crawler dispatches a new task on the dynamic crawling queue.

Responses		Errors		Crawled Hostnames	
Column	Type	Column	Type	Column	Type
Hostname	Text	*Hostname*	Text	*Hostname*	Text
Crawler	Text	*Crawler*	Text	*Crawler*	Text
URL Hash	Text	*URL Hash*	Text	At	Date
URL	Text	URL	Text		
At	Date	At	Date		
Type	Text	Message	Text		
Status	Integer				
Location	Text				
Content	Blob				
Primary Key: Hostname, Crawler, URL Hash		Primary Key: Hostname, Crawler, URL Hash		Primary Key: Hostname, Crawler	

Table 3: Data Structures: Tables

Both types of crawler store the fetched content in a persistent wide-column store. Table 3 shows the structure of the three tables used to store response content, log errors, and keep track of crawled domains. Section 3.3.3 explains why the target URL's hostname and a hash of the visited URL are part of the primary key. Along with the fetched content, the crawler saves metadata like the current timestamp, the content type, the HTTP response status code, and the final URL location after possible redirects. An example row can be seen in table 4.

Column	Key	Value
Hostname	PK	example.com
Crawler	CK	static
URL Hash	CK	a461758d8f4e0782dfa6fe66b6cdd0023c6d6894d56b4866364795c211b780f0
URL		https://example.com/login
At		2023-04-01T12:34Z
Type		text/html
Status		200
Location		https://example.com/user/login
Content		<html> …</html>

PK: Partitioning Key, CK: Clustering Key

Table 4: Responses: Example Row

3.3.3. Content Partitioning

Similarly to Bahrami et al.'s *Azure Cloud Table*-based approach [18], the response data structure in this architecture uses the hostname component of a URL as the partition key. Crawler type and URL hash are used as clustering keys[18], as shown in table 4. The partition key determines on which database instance a row will be stored, as both Cassandra and *Azure Cloud Table* use a consistent hashing ring structure to distribute partitions of content. However, an important distinction to Bahrami et al. is that the architecture in this thesis uses the target domain as the hostname, i.e., partitioning key instead of the visited URL's hostname component. This ensures all content fetched from a website is stored on the same instance, resulting in faster query times for subsequent analysis.

For instance, when crawling the URL `https://example-sso.com/login`, which was linked on the home page `https://example.com`, the partition key would be `example.com`, while the URL clustering key would be the hash representation of the visited URL.

The composition of primary keys through hostname, crawler type, and URL hash further guarantees that only one version of each response is stored. This desirable behavior prevents duplicates if an error occurs while crawling a website. Once another *worker* retries the task, duplicate content is overwritten.

However, the chosen response data structure shown in table 3 with the target domain being the partition key has a significant downside. While it allows for faster queries, the schema also allows for duplicate content if multiple target domains link to the same sign-in page or multiple websites include the same CDN-hosted JS code. For example, the domains `microsoft.com` and `xbox.com` have links to different sign-in URLs that both ultimately redirect to a URL under `login.live.com`. Another, perhaps more drastic, example would be the inclusion of common JS libraries served by CDNs, for example, at `https://code.jquery.com/jquery-3.7.0.slim.min.js`. Although, in both cases, identical content is served, the utilized data structure causes duplicates to be saved for every occurrence on a targeted domain. An alternative approach would be to split up storing content and the origin relationship. A response could be saved with the visited URL's hostname as the partition key in a *responses* table. A *links* table could then store the relation between the target domain as the origin and the final response location (after redirects). The response location would then act as a reference to the content in the *responses* table. This alternative approach would prevent storing duplicate content. It would reduce query performance in the subsequent content analysis, though. Since the storage requirements estimated in section 3.1.1 were in the range of single-digit TB values and that amount of storage is comparatively cheap nowadays, query times were prioritized when choosing the data structure shown in table 3.

[18] clustering keys in Cassandra are the equivalent of row keys in *Azure Cloud Table*

4. Implementation

After defining the overall architecture of the crawler, this section describes the practical implementation, including crawl target selection, suitable programming languages and libraries, utilized detection methods, and improving crawler stealthiness.

4.1. Target Selection

To be able to carry out content analysis, a dataset must first be composed. To collect this data, a list of target domains must be compiled to represent the web's most popular sites. At the same time, it must be clarified when a targeted domain can or must be ignored due to persisting errors or other grounds for exclusion.

4.1.1. Comparing Domain Lists

To find the most suitable list of domains representing popular sites on the web that could be used as a dataset for crawling, the following options were evaluated.

Alexa Top 1M Historically, many researchers have relied on Alexa's Top 1 Million as a basis for their studies [117, 130, 160]. While Alexa's exact methodology remained private, they used data from partnering browser extensions and website vendors [160]. In recent years, Alexa and similar lists have received some scrutiny in the literature, challenging its representativeness, stability, lack of transparency, and susceptibility to adversarial manipulations [117, 160]. Alexa has been discontinued, with their website being retired on May 1, 2022 [10]. Hence, there are no up-to-date Alexa Top 1 Million lists that even could be used for this thesis.

Majestic Million Majestic is a Search Engine Optimization (SEO) service provider publishing the *Majestic Million* popularity list that is calculated based on the number of backlinks each site has in a crawled web corpus [117, 127, 160]. Le Pochat et al. found that Majestic is especially susceptible to manipulations by adversaries by using fake backlinks to boost the rank of their website [117].

Cloudflare Radar As Alexa has reached its end of life, CDN operator Cloudflare started to publish domain rankings in its Internet statistics product *Cloudflare Radar* in 2022 [130]. Because the Internet became more centralized in recent years, a handful of companies are seeing large amounts of the overall Internet traffic [160]. Cloudflare is in the unique position of authoritatively serving traffic for about a quarter of top sites, which is significantly more than any other proviver [160]. In this context, it makes sense why Cloudflare can take advantage of its market position to produce relevant statistical data. Unfortunately, Cloudflare only publishes unordered rank buckets (e.g., top 100K, 500K, 1M) rather than a list with individually ranked domains.

Tranco Tranco is a research-oriented list of popular websites that is hardened against manipulation. It calculates rankings based on averaging multiple available rankings over a period of 30 days. With their approach, Le Pochat et al. achieve reduced fluctuation on list composition and better defense against manipulation [117]. A main advantage of Tranco is its transparent

versioning system, making research based on its lists more comprehensible and reproducible. Every new list version is archived with its generation date and a unique ID. For instance, the list that was generated on April 18, 2023, has the ID `3V6KL`. While not the case yet, Le Pochat et al. state they are working on incorporating the Chrome User Experience Report (CrUX) and Cloudflare Radar rankings in Tranco [116].

In 2022, Ruth et al. evaluated the accuracy of popular website lists by analyzing traffic data supplied by CDN provider *Cloudflare*. They found that the lists by Alexa, Majestic, Tranco, and others poorly capture web popularity. Tranco performed slightly better than Alexa and Majestic, though. Only the CrUX dataset performed notably better [160]. However, similarly to *Cloudflare Radar*, CrUX only provides rank order magnitude data (e.g., Top 1K) rather than enumerated website rank values [79, 160]. Because the volume of auditable websites within the given time constraints of this thesis was uncertain, an individually ranked list ensured continuous coverage of ranked websites for the number of ultimately crawled sites. CrUX also does not provide the benefit of making research more replicable by transparently versioning and archiving datasets. For these reasons, the Tranco list[19] [117] generated on April 18, 2023, was used for the crawl described in this thesis.

4.1.2. Handling Errors

Errors can be expected when crawling larger parts of the web. However, it is important to recognize the difference between failures that can be recovered from and ones where recovering is infeasible or impracticable. As section 3.2.1 describes, the crawler uses task acknowledgments to recover from worker failures while crawling a target. For example, sudden network dropouts or previously unconsidered states of a complex system like a remote-controlled web browser may lead to sudden failures. These rather fatal but recoverable errors result in another worker retrying after a timeout occurs and the queue redelivering the task.

If a worker detects a fatal error that is task-bound, it negatively acknowledges the task. The message broker is configured to move these tasks to a *dead queue*. After manual examination, possible bugs in the crawler's software may be corrected, and tasks may be manually moved back to the crawling queue. However, in some cases, tasks may be *poisoned* where errors will continue to occur. For example, some sites may end up in a *dead queue* because the crawling task exceeds a generous 10-minute timeout. Multi-stage timeouts are essential to prevent *poisoned* sites from clogging up workers.

Some discovered and matched URLs may not be available, perhaps because the page does not exist anymore and a `404` HTTP status code is returned, or an internal error occurs and the server responds with HTTP status code `500`. In these cases, it is impracticable to retry crawling. Instead, the requested URLs are discarded, and the next page is crawled.

In all cases, error messages are logged, and a categorized monitoring counter is increased to make accumulated errors visible in monitoring dashboards and allow for automated alerting when configured limits are exceeded.

4.1.3. Ignoring HTTP-only

One particular error that may occur repeatedly during the crawl is a failure to establish a secure TLS connection. Because a website must use TLS to be able to use the WebAuthn API [32],

[19] Available at `https://tranco-list.eu/list/3V6KL`

the whole domain is discarded if the home page cannot be reached over TLS. Since it is not technically possible for the site to utilize WebAuthn if it does not use TLS, crawling, storing, and analyzing served content would be pointless.

4.2. Choosing a Software Stack

Aside from ready-to-use system components like data stores and message brokers, custom software is needed to orchestrate and perform crawling tasks. This section discusses multiple options in regards to suitable programming languages and libraries.

4.2.1. Programming Languages

The choice of a suitable programming language is mainly determined by the following requirements. It should either have built-in functionality or a well-established library ecosystem for parsing HTML web content and interacting with the other system components, i.e., the message broker and the database. For efficiency's sake, the language should be performant to avoid wasting too many resources. To allow for concurrency within a single instance, it should also feature proper concurrency management. Because the crawler is supposed to crawl dynamic web content, the language's ecosystem should include a well-documented library for remote-controlling a Chrome browser. The language or its ecosystem should also have broad functionality for statistical data analysis. Finally, it should provide a good developer experience.

TypeScript, Python, and Go were considered viable options for this project. For all three languages, *AMQP* libraries and *Cassandra* drivers are available [19, 20, 23, 36, 88, 181]. However, the *Node.js AMQP* client has not received any updates for over two years [36]. For Python and Go, there are similar HTML parsing libraries available that can handle invalid HTML documents [105, 156], which is important when working with scraped web content. No library of similar quality and documentation coverage could be found for working with TypeScript. With *Puppeteer* and *chromedp*, the TypeScript and Go ecosystems provide libraries for remote-controlling a Chrome browser using the *DevTools Protocol* [122, 129]. For Python, no library that provided similar functionality could be found.

Regarding concurrency, Go has a clear advantage over Python and TypeScript. JavaScript (and, by extension, TypeScript) is single-threaded. *Node.js* applications achieve some level of concurrency through asynchronous callbacks and an event loop. Using multiple cores per machine would require running multiple crawler instances per machine, each with a single-threaded event loop. And while Python supports a form of threading, no real concurrency can be achieved. Threads may run on different processors, but not at the same time. In contrast, Go features *goroutines*, which can be considered lightweight threads that are managed by the Go runtime. Because of the low cost of creating a goroutine, applications may use numerous *goroutines* simultaneously. That is unlike languages using traditional threads, where it is typically recommended to create threads equal (or double) the number of processing cores. Thanks to a well-equipped standard library, good readability, and efficient garbage collection, Go provides a good developer experience and excellent performance while being memory-safe.

Since Go best meets the requirements, the language was used for the development of the crawler. For the last step – statistical evaluation of the analysis results – Python was used due to its vibrant ecosystem in data processing and visualization.

4.2.2. Suitable Libraries

Because of its maturity and adequate level of documentation, the *gocql* library [20] is used as a Cassandra driver. As the RabbitMQ core team maintains a first-party feature-rich and well-documented RabbitMQ client for Go [181], it is used to communicate with the message broker.

To parse crawled web content, the *soup* package [105] is used. *Soup* provides an interface highly similar to the popular *BeautifulSoup* Python library. Like its role model, *soup* is able to parse invalid and partly broken HTML documents while allowing for straightforward DOM-traversing queries. Listing 7 shows a simplified example of querying for HTML inputs that indicate *Conditional UI* support.

Listing 7: Simplified Soup Query Example

```
root := soup.HTMLParse(content)
if root.Error != nil {
    // Handle error
}

for _, node := range root.FindAll("input") {
    if value, ok := node.Attrs()["autocomplete"];
        ok && strings.Contains(value, "webauthn") {
        // Input with conditional UI reference found
    }
}
```

Lastly, *chromedp* is used to drive a web browser using the *DevTools Protocol* [129]. Large parts of its interface are autogenerated from the *DevTools Protocol*, allowing for complete feature coverage. Using *Chromdp*, Go applications can do everything a developer could do manually in their browser's development tools. That includes performing actions like opening a new tab, navigating to a website, pressing a button, or filling a text input field. It also includes listening to browser events. For example, a tab's complete network activity can be intercepted, allowing the crawler to pause, analyze, and continue or cancel specific HTTP requests. If a suitable browser executable is present in the deployment environment, *chromedp* can start and stop browser instances as needed.

4.3. Detection Methods

The crawler needs to detect two different things: authentication URLs and utilized authentication technologies. The former is needed to identify relevant URLs on the home page of targeted sites. The latter is needed to answer R1.

4.3.1. Authentication URL Detection

Because the crawler is scoped, it is crucial to identify possibly relevant pages. This crawler's approach is to match hyperlinks on home pages that could refer to secondary pages with authenticating capabilities. For instance, a matched URL may serve a HTML sign-in form. By exclusively crawling those matched URLs while ignoring irrelevant content, the crawler is able to save on storage requirements and minimize crawl time.

There are two simplistic approaches to detecting a sign-in hyperlink: matching the referenced URL and matching the anchor tag's visible text label. It is important to detect the page's

language so that a localized list of patterns can match any anchor tag's text label. Otherwise, large parts of the non-English speaking web would be strongly underrepresented. Fortunately, valid HTML documents must include a `lang` attribute representing the document's utilized language. The corresponding language tags must adhere to the format defined in RFC 5646. For example, `fr`, `en-US`, and `de-DE` are all valid language tags identifying language and, in some cases, region [144]. Unfortunately, significant fractions of the web do not serve valid HTML documents. The 2022 *Web Almanac* report shows that 18 % of sites on a desktop did not have a language set at all [132], which makes language detection difficult. Nonetheless, the crawler uses a localized list of keywords for the most popular languages on the web identified by the *Web Almanac* report [132]. For the most part, these keywords are automatically translated using the results from *DeepL*[20] and *Google Translate*[21] cumulatively. If no language tag is set, the English keywords are used for matching.

The second approach for identifying relevant anchor tags is to match their referenced URL based on their path. Listing 8 shows a list of regular expressions the crawler uses to match discovered URLs. To prevent undesirable content, some patterns exist that exclude URLs when they match. This prevents crawling large files like PDF documents, images, videos, or office documents (line 17). It also prevents the crawler from re-visiting the same page if URLs contain a fragment component referencing content within the page (line 16, e.g., `https://example.com/#about-us`). To prevent unwanted behavior, only relative URLs or absolute URLs having the `http` or `https` scheme are processed. Anything else is discarded, for example, opaque data with a `data` scheme or scripts with a `javascript` scheme.

Listing 8: Including and Excluding Authentication URL Patterns

```
1  func (m authURLMatcher) matchURLPatterns() []string {
2      return []string{
3          `log-?in(\W|$)`,
4          `auth(enticate)?(\W|$)`,
5          `register(\W|$)`,
6          `registration(\W|$)`,
7          `account(\W|$)`,
8          `sign-?(in|up)(\W|$)`,
9          `admin(\W|$)`,
10     }
11 }
12
13 func (m authURLMatcher) excludeURLPatterns() []string {
14     return []string{
15         `^#.*`,
16         `.+\.(pdf|jpg|png|gif|psd|heic|docx?|xlsx?|csv|pptx?|aif|flac|m3u|m4a|mp3|ogg|wav|wma|avi|flv|m4v|mov|mp4|ts|vob)$`,
17     }
18 }
```

An alternate approach to discovering authentication URLs is to match them in a website's `robots.txt` file. As section 2.2.3 describes, the *Robots Exclusion Protocol* defined in RFC 9309 governs how website operators can explicitly specify the paths they want to allow or disallow specific robotic visitors to fetch [111]. Part of an ordinary politeness policy usually is to honor these wishes. However, the use case of the described crawler entails crawling content that is

[20] `https://www.deepl.com`
[21] `https://translate.google.com`

typically not supposed to be indexed, for instance, by search engines. After careful consideration, this thesis considers it justifiable to crawl pages explicitly disallowed by a website's `robots.txt` configuration in violation of RFC 9309 due to the crawler's narrow scope and slowness, resulting in minimal load.

4.3.2. Authentication Method Detection

Detecting the use of authentication technologies within fetched content is paramount for answering R1. At the same time, it is a hard problem to solve. Identifying all uses of a specific API would be impossible, thus the recognition rate will never be 100 %. The first major problem is detecting whether a client-side rendered website, essentially being a black box, has finished, or will eventually finish, loading additional JS resources. This is a classic instance of the *halting problem*, which is undecidable.

The second problem is identifying the use of a specific API within a decision tree of unknown breadth and depth. One could try to trigger every possible combination of events to provoke a direct API call. However, the resulting effort would be enormous, slowing down crawl times significantly while increasing load drastically. And even then, some resources may only be loaded, and some actions may only be performed when a website's backend triggers a server-side event. For example, in an MFA flow where the user initially authenticates with an email address and a password before being prompted for a FIDO credential as a second factor, loading the respective client-side JS code may be deferred until it is necessary.

Furthermore, JS code is frequently minified and/or obfuscated, making static analysis hard. Listing 9 shows a few simple examples of how WebAuthn API calls may be obfuscated. Of course, much more advanced obfuscations are conceivable.

Listing 9: Examples of JavaScript Obfuscation

```
1  // Typical WebAuthn API call
2  navigator.credentials.get({...})
3
4  // Unsophisticated obfuscation
5  navigator['credentials'].get({...}
6
7  // Somewhat sophisticated obfuscation (formatted for readability)
8  let d = function(a, b, c) {
9    a[atob(c) + 't' + b].get({...});
10 }
11 d(navigator, 'ials', 'Y3JlZGVu'); // = base64('creden')
```

To improve detection rates, one solution could be de-obfuscation and string reconstruction similar to the *Restringer* project [21]. However, this approach requires considerable effort and will likely not work in every case. Therefore, the crawler's content analysis is limited to simple matching techniques. They could still, for instance, detect the WebAuthn call in line 5 in listing 9, but are not sophisticated enough to reconstruct strings or reverse deliberate obfuscation tactics.

Before analyzing content, the crawler splits all fetched content for a website by content type and applies a different set of matching rules. For HTML content, the crawler also extracts any inline JS code that may be embedded. Specifically, the crawler is able to recognize authentication methods using the following ruleset that is summarized in table 5.

- Usage of the `isConditionalMediationAvailable()` static method of the *PublicKeyCredential* interface is detected in JS code. If conditional mediation is available, users can autofill discovered FIDO credentials through the browser's non-modal *Conditional UI* dialog [131, 166]. Listing 4 contains a detectable example. Usage is evidence that the website allows passkeys, i.e., discoverable multi-device FIDO credentials for authentication. While it is not detectable whether additional authentication factors are required, usage of this method is a good indicator that the website may support *passwordless* single-factor FIDO-based authentication.
- In the corresponding HTML, sites may use the `autocomplete="webauthn"` attribute on inputs signaling the web browser should display a *Conditional UI* dialog if the user focuses on the input [166]. Listing 5 contains a detectable example. Usage is detected by traversing the DOM tree and matching any inputs with corresponding `autocomplete` attribute values.
- While the previous methods detect capabilities that were only recently added to the WebAuthn specification with the introduction of *passkeys* [101], a straightforward detection method is to match calls to the `navigator.credentials` WebAuthn JS API. By itself, usage of the API is only evidence that some form of FIDO credential can be used to authenticate. This may be as a second factor within a MFA flow or as a single *passwordless* factor, though. However, the rule is able to detect static use of the *conditional mediation* request option, indicating that the site employs *passkey* authentication.
- WebAuthn's predecessor, *U2F*, is detected in JS by matching commonly used library imports and corresponding library API calls. For example, a site's JS code may include `require('u2f-api')` or `window.u2f.sign()`, which would suggest they still support *U2F*.
- Finally, traditional password-based sign-in forms are discovered by traversing the HTML's DOM tree and matching any inputs that have a `password` type or whose `name` attribute suggests it is a password input.

Rule	Technique	Content Type	Detectable Methods
Conditional Mediation Available	Regex-based	JS	Passkeys
Conditional UI	DOM-based	HTML	Passkeys
Navigator Credentials API	Regex-based	JS	WebAuthn, Passkeys
U2F	Regex-based	JS	U2F
Password Input	DOM-based	HTML	Passwords

Table 5: Implemented Authentication Method Matching Rules

4.4. Preliminary Experiments

Before carrying out a real-world, large-scale crawl, several small-scale tests were performed to find potential for optimizations, catch any programming errors, and test hypotheses.

4.4.1. Unit Testing with Real Web Content

To ensure proper matching behavior, the URL and authentication technology matching components are covered with several unit tests. While some of these tests use arbitrary examples to

check that the code behaves as intended, others verify that matching works on real web content. For URL matching, 62 extracts were gathered from samples of the top 200 domains on the *Tranco* list described in section 4.1.1. The samples include *Adobe*, *Dropbox*, *Ebay*, *Microsoft*, *Netflix*, *Reddit*, *The Guardian*, *Twitter*, *Wikipedia*, and *YouTube*. These real-world HTML extracts also help to ensure localized matching with automatically translated labels works as intended. For instance, test cases for *mail.ru*, *qq.com*, and *baidu.com* verify that the extracted text labels in Russian and traditional Chinese are matched correctly.

4.4.2. Sitemap Authentication URL Extraction

In 2010, Olston and Najork listed the collaboration between content providers and crawlers as a relevant future field of crawler research, mentioning that many sites use the *Sitemap Protocol* to publish a machine-readable list of available content [142, 169]. To increase coverage of authentication pages, fetching sitemaps and matching referenced URLs sounds promising.

However, after manually inspecting sitemaps from a sample of the top 200 domains on the *Tranco* list described in section 4.1.1, it became clear that most websites only publish URLs referencing content they want to see indexed by search engines, for example, blog articles or product pages. Unfortunately, URLs to sign-in forms were rarely found in the sampled sitemaps. Thus, sitemap crawling was omitted in favor of other, more promising approaches.

4.4.3. Optimizing Chrome Crawling Performance

Compared to the static crawler using a simple HTTP client included in Go's standard library, crawling using a Chrome browser carries a significant performance penalty. However, the dynamic crawler implementation uses several methods to optimize crawling performance and improve task throughput.

Reusing Instances and Its Tradeoffs

Typically, it is desirable to crawl a website without any preexisting state that could influence the resulting content. Simply put, every task should be started with a clean slate. However, crawling dynamic, client-side rendered content using a web browser like Chrome brings significant performance penalties when a fresh browser instance is started for every crawling task. Starting, stopping, and clean-up phases add up to multiple seconds per occurrence. Crawling every individual page with a fresh Chrome instance would therefore be unfeasible. Even starting a fresh instance per crawling task (targeted website) is way too cost-intensive.

When a browser process is reused for multiple crawls, there are two potential negative effects: previous state affecting future content and disk clogging or memory leaks. When a browser collects browsing history, most visited websites store some state within cookies. They may also persist data in the browser's local or session storage. This previous state can influence servers to behave differently, for example serving altered or alternate content compared to the response a new visitor would receive. Moreover, using a browser instance over a prolonged period of time may cause the disk to be clogged by temporary files that are not cleaned up after a page visit. As a web browser is a fairly complex software, it may also suffer memory leaks that fill up Random Access Memory (RAM).

Ultimately, these tradeoffs must be accepted if the alternative of starting a fresh browser instance for every request or task is not feasible. While disk space leakage can be handled, as

section 7.2.1 discusses, possible content changes due to previous state are not traceable and cannot be prevented, although it is estimated that the consequences are only minor regarding this work's use case.

Tabs vs. Windows

While reusing a Chrome instance for a prolonged period of time provides significant benefits in crawling performance, further optimizations need to be made. While a static crawler instance can use many *goroutines* to start workers and fetch content concurrently, a dynamic crawler instance would need exorbitant amounts of RAM and Central Processing Unit (CPU) cores to host individual Chrome instances per worker *goroutine*. Fortunately, *chromedp* allows concurrent use of a single browser instance by multiple worker *goroutines*. This begs the question of the best way for workers to use one Chrome instance concurrently: Should they open separate windows or tabs within a single window?

Chrome is a multi-process application that uses a *Site Isolation* architecture to protect users' privacy and reduce the attack surface for transient execution attacks like *Spectre* and *Meltdown* [154]. The architecture effectively provides a sandbox for every site by locking each renderer process to its own documents [154]. Hence, not just every window but every tab visiting a separate site is rendered within its own process isolated by the operating system [154, 155]. One would assume that sandboxing does not entail any differences in the amount of separate processes.

	100× Same Site		Tranco Top 100	
	Tabs	Windows	Tabs	Windows
Memory usage	13.38 MB	42.21 MB	10.93 MB	39.91 MB
# Processes	87	802	90	770

Table 6: Chrome Tabs vs. Windows Measurement Results

However, experimental measurements suggest a difference in memory usage and process count between using multiple tabs and multiple windows. Two separate experiments were performed to study the differences *Site Isolation* may introduce. In the first experiment, one site[22] was concurrently visited 100× using multiple tabs and then multiple windows. In the second experiment, the top 100 sites on the Tranco list[23] [117] generated on February 22, 2023, were concurrently visited using multiple tabs and subsequently multiple windows. Both experiments were conducted on a machine with an *Apple M1 Pro* System on a Chip (SoC) and 32 GB of available RAM running macOS Ventura and Chrome 110.0. Table 6 shows that using windows instead of tabs resulted in over three times the memory pressure in both experiments. The number of running Chrome processes increased by a factor of ≈ 9 when using windows. In a nutshell, while the effects of *Site Isolation* are negligible, using tabs for concurrent crawling provides significantly better performance.

[22] `https://www.hdm-stuttgart.de`
[23] Available at `https://tranco-list.eu/list/LYVG4`

Blocking Media To Save Bandwidth

Another utilized measure to optimize crawling performance is intercepting a page's network traffic and pausing all fetch requests. This allows the crawler to analyze outgoing HTTP requests before the web browser sends them to the network. Listing 10 shows a simplified code excerpt of how the crawler uses this capability to block all HTTP requests for undesired content like images, videos, font files, or stylesheets. Thereby, unnecessary network traffic is prevented, which allows for higher throughput of relevant content.

Listing 10: Simplified Media Blocking Code Excerpt

```
1  func (c Crawler) reviewRequest(ctx context.Context, ev *fetch.EventRequestPaused) {
2          ...
3
4          blockedResourceTypes := []network.ResourceType{
5                  network.ResourceTypeImage,
6                  network.ResourceTypeMedia,
7                  network.ResourceTypeFont,
8                  network.ResourceTypeStylesheet,
9          }
10
11         if slices.Contains(blockedResourceTypes, ev.ResourceType) {
12                 return fetch.FailRequest(ev.RequestID, network.
       ↪ErrorReasonBlockedByClient).Do(ctx)
13         }
14
15         return fetch.ContinueRequest(ev.RequestID).Do(ctx)
16 }
```

4.5. Avoiding Crawler Detection

For several years now, an arms race is taking place between crawler and website operators – and both sides have valid arguments. On the one hand, web crawling is essential for many businesses and researchers alike. On the other hand, some crawlers act inconsiderately or even in bad faith. For instance, in 2023, aggressively scraping large amounts of data provided by the non-profit *Internet Archive* has repeatedly interrupted their availability [102]. Many businesses also face problems like stolen, unattributed content or malicious attacks like spamming or phishing that take place over the web. While there are valid reasons for trying to prevent undesired bot interactions, *overblocking* negatively affects legitimate use cases like Internet research.

Constantly, new bot detection techniques are being deployed and crawling operators try to find circumvention methods. For example, some sites may try to detect automated visits by analyzing a client's user agent, permissions (e.g., for notifications), list of enabled plugins, supported languages, or its ability to draw an image on a canvas [163, 164]. Of course, workarounds for all of the mentioned detection techniques already exist [163]. A somewhat popular bot detection test is available at `https://bot.sannysoft.com`. The site creates a report on a client's performance in various detection scenarios. For instance, in 2018, Sangeline used it to demonstrate how all included tests can be circumvented by injecting some JS code into the browser before visiting the testing site [163].

Optimizing Chrome Stealthiness

Fortunately, a stealth plugin for the *Node.js* library *Puppeteer*, which is mentioned in section 4.2.1, exists that successfully evades all publicly available bot tests by injecting JS code into the browser before a page visit [83]. This code patches several browser APIs to produce unsuspicious outputs. For example, the code leads website operators to believe a particular set of plugins was installed in the browser by creating mock functions and patching the respective browser API [83].

The evasion rules made available by the *Puppeteer* stealth plugin [83] are compiled and used to implement a dynamic crawler stealth mode, optionally configurable using a feature flag. Figure 3 shows an excerpt of the bot test results for the dynamic crawler running on a Debian Virtual Machine (VM) in the final *OpenStack*-based deployment environment. The machine does not have a graphical UI, and Chrome is operated in *Headless* mode, so naturally fig. 3a features some suspicious test results. With enabled stealth mode, Figure 3a exhibits an improved behavior for the *WebDriver* and *WebGL Renderer* tests. Note that the missing *Hairline Feature* result also occurs in "headful" Chrome, and its absence is not critical.

	Result
User Agent (Old)	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.0.0 Safari/537.36
WebDriver (New)	present (failed)
WebDriver Advanced	passed
Chrome (New)	present (passed)
Permissions (New)	prompt
Plugins Length (Old)	5
Plugins is of type PluginArray	passed
Languages (Old)	en-US,en
WebGL Vendor	Google Inc. (Google)
WebGL Renderer	ANGLE (Google, Vulkan 1.3.0 (SwiftShader Device (Subzero) (0x0000C0DE)), SwiftShader driver)
Hairline Feature	missing
Broken Image Dimensions	

(a) Stealth Options Disabled

	Result
User Agent (Old)	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.0.0 Safari/537.36
WebDriver (New)	missing (passed)
WebDriver Advanced	passed
Chrome (New)	present (passed)
Permissions (New)	prompt
Plugins Length (Old)	5
Plugins is of type PluginArray	passed
Languages (Old)	en-US,en
WebGL Vendor	Intel Inc.
WebGL Renderer	Intel Iris OpenGL Engine
Hairline Feature	missing
Broken Image Dimensions	

(b) Stealth Options Enabled

Figure 3: Results of Sannysoft Bot Detection Test[24]

New Chrome Headless Mode

When Chrome first introduced a *Headless* mode, which lets developers run the browser unattended without any visible UI, it ran on an entirely different implementation. "Headful" and *Headless* Chrome did not share any code, which resulted in differing behavior, separate environments where bugs may occur, and additional maintenance overhead [26, 41]. In 2023, Chrome introduced a new *Headless* mode that shares a code path with the "headful" Chrome browser. This guarantees uniform behavior and the availability of all existing and future functionality in *Headless* mode [41].

In combination, by injecting stealthiness JS code into the Chrome browser and using the new *Headless* mode, which shares the same code with its "headful" counterpart, the dynamic crawler is supposed to appear to site operators as much as possible like an ordinary visitor. Of course, this is a cat-and-mouse game where sophisticated operators such as prominent CDN vendors may still detect and block automatic crawling through novel techniques enabled by unique market positions, allowing them to automatically analyze large parts of all Internet traffic.

[24] Available at `https://bot.sannysoft.com`

5. Deployment

This section describes how the implemented web crawler was deployed into an *OpenStack* environment. It also discusses some workarounds that had to be used due to the peculiarities of the deployment environment. Furthermore, the deployed storage and queuing components are load tested to determine the proper resource sizing. Finally, the optimization evolution of deployed crawler instance specifications is reviewed.

5.1. Automated Deployment

Creating a fully-automated and reproducible deployment environment is achieved through a combination of cloud infrastructure provisioning using *Terraform* [86] and deployment automation using *Ansible* [48]. Because only common open-source cloud components are used, the deployment is theoretically adaptable to any cloud environment without any vendor lock-in. As part of this thesis, the distributed crawler was deployed to the OpenStack-based *bwCloud*[25], which is a state-run Infrastructure as a Service (IaaS) provider for public science and education institutions within the German federal state Baden-Württemberg. While it is commendable that the state provides this service, Sections 5.3, 6 and 7.2.1 discuss several problems that specifically arose as a result of relying on *bwCloud*.

Terraform

Terraform is used to provision the needed infrastructure through *bwCloud*'s OpenStack APIs. The Terraform modules create block storage resources, respective attachments, security groups, and compute resources of configurable amount and flavor, i.e., available combinations of VM specifications like CPU cores and available RAM. Throughout the crawling process, this flexibility allowed continuous adjusting of the ratio of dynamic and static crawler instances to optimize for similar throughput rates. Section 6.1 covers this in detail.

The fact that *bwCloud* only supports block storage but currently does not provide any object storage also influenced the decision to store crawled content entirely in a wide-column store addressed in section 3.2.2. Otherwise, deploying a custom object storage service on dedicated compute and block storage resources would have been necessary.

Aside from provisioning infrastructural components, Terraform is also used to create inventory files of the created compute resources for subsequent configuration using Ansible. This way, the integration of both tools is seamless, and a complete provisioning and deployment process can be fully automatable.

Ansible

After provisioning VMs with a Debian 11 image using Terraform, an *Ansible playbook* is executed on the hosts listed in the generated inventory file. Different groups separate the hosts so that separate Ansible roles can be applied to each host group. Every deployed capability is configured within a dedicated Ansible role, so some configurations can be applied to all hosts, while others only apply to a specific group. For example, utilities like the network speed measurement tool `iperf3` and a monitoring data exporter are installed on every host, while the Chrome package is only installed on dynamic crawler instances. Ansible is also used to mount the provisioned

[25] `https://www.bw-cloud.org`

block storage on hosts that need to store persistent data. Because the Terraform and Ansible configurations are both designed to be idempotent, changes can be made to the deployment without needing to recreate a fresh environment.

Overall, Terraform and Ansible are used to deploy a Cassandra cluster, a RabbitMQ instance, several static and dynamic crawler instances, and a dedicated monitoring instance. The instances hosting Cassandra, RabbitMQ, and a monitoring stack also have persistent block storage available. Due to the peculiarities of the deployment environment discussed in section 5.3, a *NAT64* gateway in a separate location is deployed additionally to ensure the crawlers have proper dual-stack connectivity.

5.2. Monitoring

To monitor all relevant software components within the deployment, a combination of *Prometheus* and *Grafana* is used [140, 153]. Prometheus collects various metrics from exporting services on the monitored nodes by scraping their HTTP endpoints in intervals of 15 s. To instrument the VMs, the first-party `node-exporter` program is installed directly on the hosts. In terms of architectural components, there are differing approaches for instrumentation.

While RabbitMQ comes with a plugin that directly exposes a dedicated HTTP endpoint for Prometheus metrics, monitoring Cassandra is slightly more complex. Since Cassandra is built using Java, it relies on instrumentation via Java Management Extensions (JMX). Fortunately, the community around Prometheus has created *JMX Exporter* [35], which is a collector that can scrape JMX *Bean* targets. However, proper integration and configuration are not well-documented and, at least in this particular instance, was somewhat challenging.

For the developed crawler software, custom metrics are collected using the Prometheus Go client [150]. They are exposed on a dedicated HTTP endpoint with Go's standard library HTTP server. To allow for application-specific instrumentation, Prometheus collects metrics like the total number of jobs per type of crawler (static or dynamic). It also collects counters for errors, outgoing HTTP requests, incoming HTTP responses, and respective HTTP status codes. Furthermore, it measures the duration of database and queue operations and outgoing HTTP requests. The number of discovered authentication URLs per website and the number of currently active workers are tracked, too.

For every aforementioned metrics exporter, a dedicated HTTP endpoint allows employing security measures at a network level by restricting connections by their source IP address. In this case, only the monitoring host is allowed to establish a Transmission Control Protocol (TCP) connection to the metrics endpoint, while any other connection attempts are discarded.

5.3. Peculiarities of the Deployment Environment

Relying on *bwCloud* to deploy the crawler resulted in several problems. This section outlines issues caused by Internet Protocol version 4 (IPv4) address space exhaustion affecting the region of the assigned OpenStack tenant. It also describes methods used to overcome these issues.

5.3.1. IPv6-Only Connectivity

As the available IPv4 address space within in the assigned *bwCloud* location Mannheim is currently exhausted, all hosts within the deployment only had Internet connectivity through Internet Protocol version 6 (IPv6). This was problematic because large parts of the Internet still

rely on IPv4-only connectivity. If the web is to be surveyed, a crawler must be able to connect to those web servers only available via IPv4. For instance, at the time of writing, *Stuttgart Media University*'s entire network cannot route any IPv6 packets, rendering communication between the deployment environment and the university network impossible. To solve this issue, a *NAT64* gateway detailed in section 5.3.2 was deployed in another *bwCloud* location. The gateway provided a tunnel for communicating with IPv4-only hosts.

What worsened the situation was that hosts in *bwCloud*'s Mannheim region actually had dual-stack networking. However, the private class A IPv4 network (`10.0.0.0/8`) did not route any packets to the Internet. This caused any software preferring IPv4 connections to be unable to communicate with hosts on the Internet. After discovering this behavior repeatedly, the removal of all IPv4 routes from the provisioned hosts was ultimately added to the Ansible deployment as part of the necessary mitigation.

Unfortunately, IPv6 still does not enjoy the same level of application support as IPv4. For example, *Docker*'s IPv6 support is still experimental [57]. This is one of the reasons why containerization was not used in this particular deployment, although all components can be containerized. While some software like RabbitMQ requires additional configuration effort to enable IPv6, the crawler's Java-based software components Cassandra and *JMX Exporter* needed to be actively prevented from preferring the IPv4 stack and ignoring their IPv6 connectivity after IPv6 was already configured.

5.3.2. NAT64 Gateway

A NAT64 gateway facilitates communication between IPv6 and IPv4 hosts by using a form of Network Address Translation (NAT). To allow hosts in the Mannheim deployment region to communicate to IPv4 hosts on the Internet, the out-of-kernel stateless NAT64 implementation *TAYGA* was used [123]. It creates a tunnel network for each IP version and acts as a router within both networks, translating destination IP addresses back and forth. IPv6 hosts can route packets to their final IPv4 target by addressing it to a translated address within the well-known `64:ff9b::/96` NAT64 prefix. The IP packet needs to be routed to the NAT64 gateway, which translates the destination address to IPv4 and routes the packet to its next hop on an IPv4 network. In fact, the IPv6 host does not need to know about the existence of the IPv4 address at all.

Instead, when hosts try to resolve hostnames using DNS, they should only receive `AAAA` records with IPv6 addresses, even when a hostname only has an `A` record available. A DNS server can accomplish this using DNS64. In the described deployment, the widely-used DNS server *BIND 9* [96] is configured to respond to hostname queries with `AAAA` records, even when none originally exist. In such a case, *BIND 9* is configured to translate the IPv4 addresses from any `A` records into IPv6 addresses within the well-known `64:ff9b::/96` NAT64 network and serve them as `AAAA` records.

All hosts in the Mannheim deployment region are configured to route packets with a destination address within the well-known NAT64 network to the NAT64 gateway located in *bwCloud*'s Ulm region. This behavior is set using a Wireguard Virtual Private Network (VPN) configuration, which is required for secure communication anyway.

Unfortunately, communication over a NAT64 gateway is limited by the gateway's maximum throughput. To assess the negative performance impact of NAT64 tunneling, a speed test was performed using an *iPerf* [61] client. The 10 Gbit/s *iPerf* server publicly available at

`ping.online.net` and `ping6.online.net` was used as a counterpart in both tests. All provisioned hosts in both *bwCloud* regions have a theoretical network connection of 10 Gbit/s. When performing an *iPerf* test over IPv6, the data rate was ≈ 950 Mbit/s in both directions, which roughly corresponds to the theoretical maximum. When running *iPerf* over the NAT64 tunnel, the data rate in both directions is reduced to ≈ 365 Mbit/s. That is only 38 % of the original throughput. Of course, the theoretical data rate maximum is half of the gateway's network connection. Any additional reduction is presumably caused by the involved address translation overhead. While this significant performance penalty is indeed unfortunate, using NAT64 is the only viable solution to allow connections to IPv4 hosts in this particular deployment environment.

Because it became necessary to explore IPv6 address translation, an additional metric was added to the crawler, which collected data on the adoption of IPv6 among visited sites. Section 6.1.1 discusses the observed results.

5.3.3. Mesh Virtual Private Network

Because *bwCloud* provided an OpenStack tenant without the ability to create private networks or network routers, which are basic functionalities typically provided by one of OpenStack's core components, a VPN was used to provide a secure communication tunnel between deployed hosts. Wireguard [58] was used to create a private mesh VPN because its peer-to-peer connectivity and adequate performance were desirable.

At the provisioning stage, Terraform assigns an incrementing Unique Local Address (ULA) in the address range `fc00::/7` to every compute resource and outputs it into the host inventory files parsed by Ansible. At the deployment stage, Ansible creates an asymmetrical key pair for every possible host pair combination. It also creates pre-shared symmetrical keys for every host pair for additional security. If matching keys exist on the target machine, key generation is skipped to ensure idempotence. Ansible utilizes a *Jinja* template to produce a unique Wireguard configuration for each VM in the deployment. This includes registering all peers, generated keys, the assigned ULA, and the corresponding Wireguard endpoint. The endpoint consists of the peer's public IPv6 address and the User Datagram Protocol (UDP) port where the Wireguard client is listening. This setup allows for secure communication through a private, tunneled IPv6 network.

The Wireguard configuration specifies the NAT64 gateway peer's ULA as the next routing hop for the `64:ff9b::/96` NAT64 network. It also configures all hosts to use the NAT64 gateway as a DNS server, where *BIND 9* provides DNS64 services.

5.4. Load Testing Components

The assigned *bwCloud* OpenStack tenant has a quota of 20 compute instances, a total of 80 virtual Central Processing Units (vCPUs), 120 GB of RAM, and 3 TiB of available block storage. To ensure the optimal distribution of available resources, several load tests were performed on the internal crawler components. Specifically, sufficient capacities for Cassandra and RabbitMQ needed to be provided, while as many resources as possible should remain available for the actual crawler instances. Moreover, due to the limited number of available instance *flavors*, which determine a VM's vCPUs and RAM combination, not all quotas could be used to their full potential.

The regular dynamic and static crawler instance deployment is used to provision load test clients to make the load tests as realistic as possible. Thus, the same number of instances with the same resource limits will also be used in the crawling process.

5.4.1. Cassandra

The load tests were designed to determine whether it would be better to provision three smaller Cassandra instances that divide their workload equally or a single large instance that eliminates cluster communication overhead. However, all instances had a network connection limited to 1 Gbit/s, which might be insufficient when using a single Cassandra instance.

During each load test, each concurrent worker of a client instance repeatedly stores exemplary HTML documents. The configured number of concurrent workers per client instance is increased over time with the goal of overstraining the Cassandra instances.

Load Test 1

The first load test used three Cassandra instances with 4 vCPUs and 8 GB of RAM each. Table 7 details the remaining instance specifications.

Name	Instances	Flavor	vCPUs	RAM
Cassandra	3	m1.large	4	8 GB
Dynamic Crawler	6	m1.large	4	8 GB
Static Crawler	7	m1.medium	2	4 GB

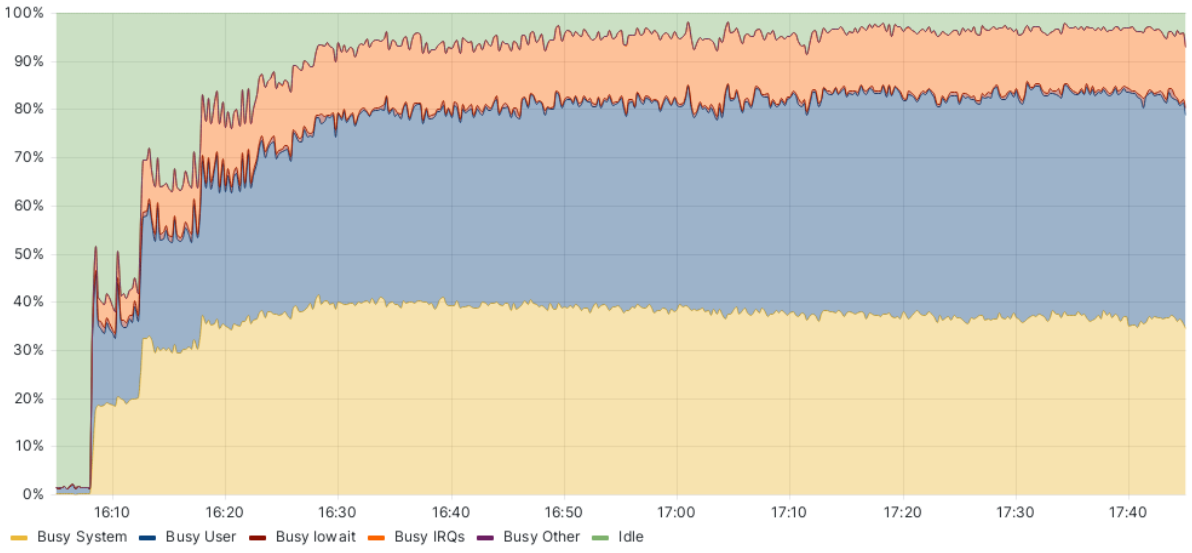
Table 7: Instance Specifications for Cassandra Load Test 1

During the first load test, the number of concurrent *goroutines* on each of the 14 client instances was gradually increased using the following steps: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 20, 25, 30, 35, 40, 50, 70, 100, 150, 200, and 400. Figure 4a shows that the CPU usage on all three Cassandra instances increased fairly quickly, plateauing at $\approx 95\,\%$ after reaching ten concurrent workers per client instance (16:30 in fig. 4a). However, no errors occurred until 400 concurrent workers per instance were reached. Figure 4d shows that the duration of storing operations increased roughly linearly until the number of workers per instance reached 200.

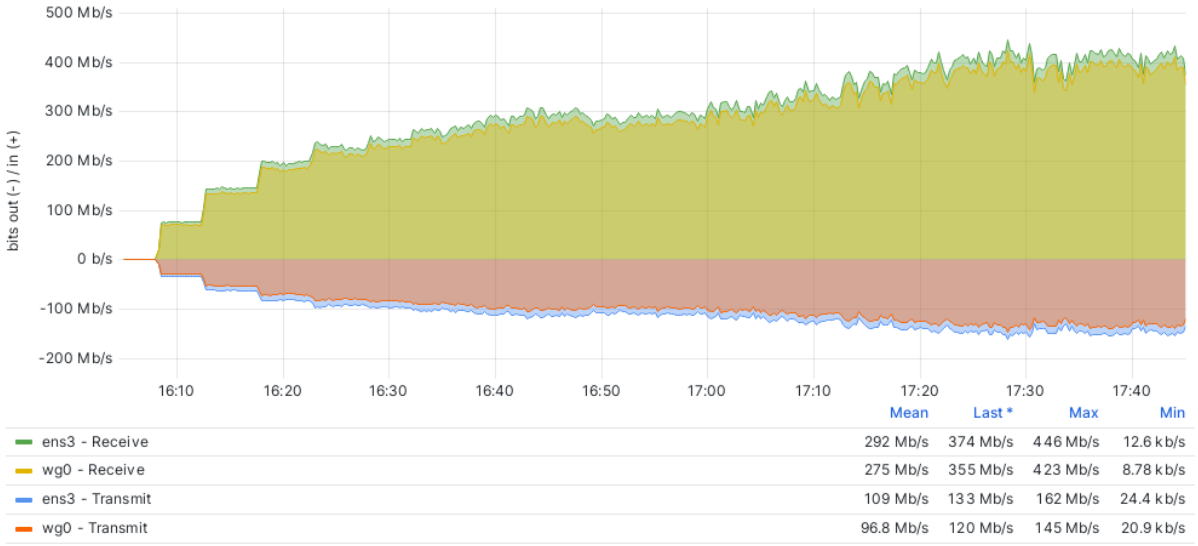
After that point, the duration more than doubled. At the same time, the number of Cassandra's pending tasks displayed in fig. 4c started to increase noticeably. Finally, at 400 workers per instance, the duration of store operations increased drastically from ≈ 250 ms to ≈ 880 ms per operation. The pending tasks increased exponentially, peaking at over 300. Workers started to receive errors when trying to insert records at that time. Hence, the number of 400 concurrent workers per instance can be regarded as the threshold at which the cluster of three Cassandra instances with an *m1.large* flavor is overstrained.

Figure 4b shows the network traffic on the first Cassandra instance, which peaked at 446 Mbit/s. The other two instances had maximum throughputs of 428 Mbit/s and 309 Mbit/s. In purely arithmetical terms, a cumulative maximum of 1,183 Mbit/s would mean that a single, larger-sized Cassandra instance had insufficient bandwidth, with the network link potentially becoming

a bottleneck. Yet, this may be compensated by the removal of possible communication overhead between Cassandra instances.

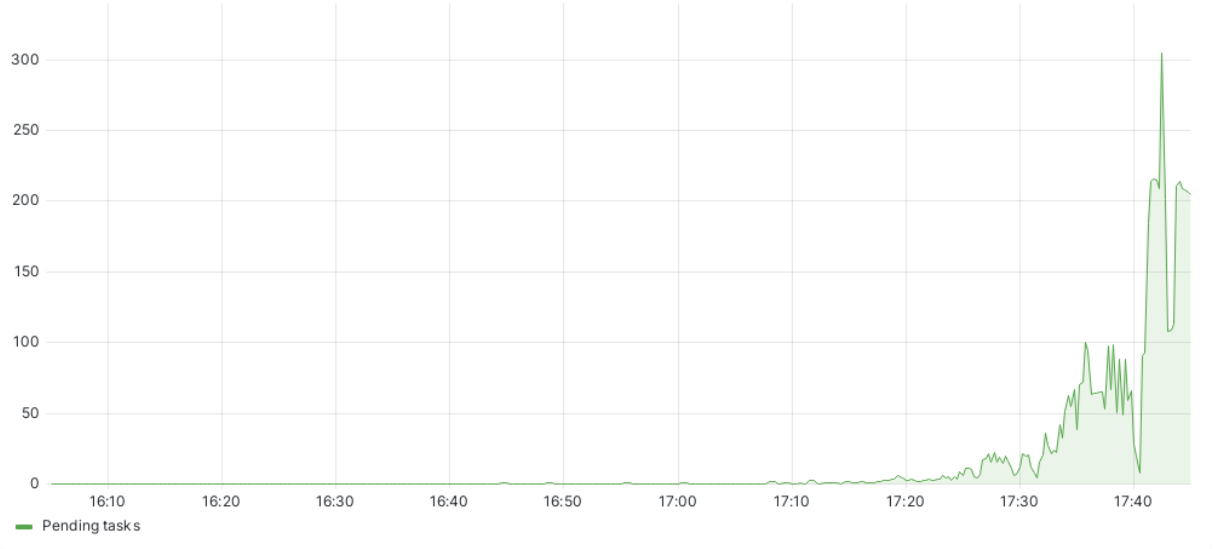


(a) CPU Usage – `cassandra-1`

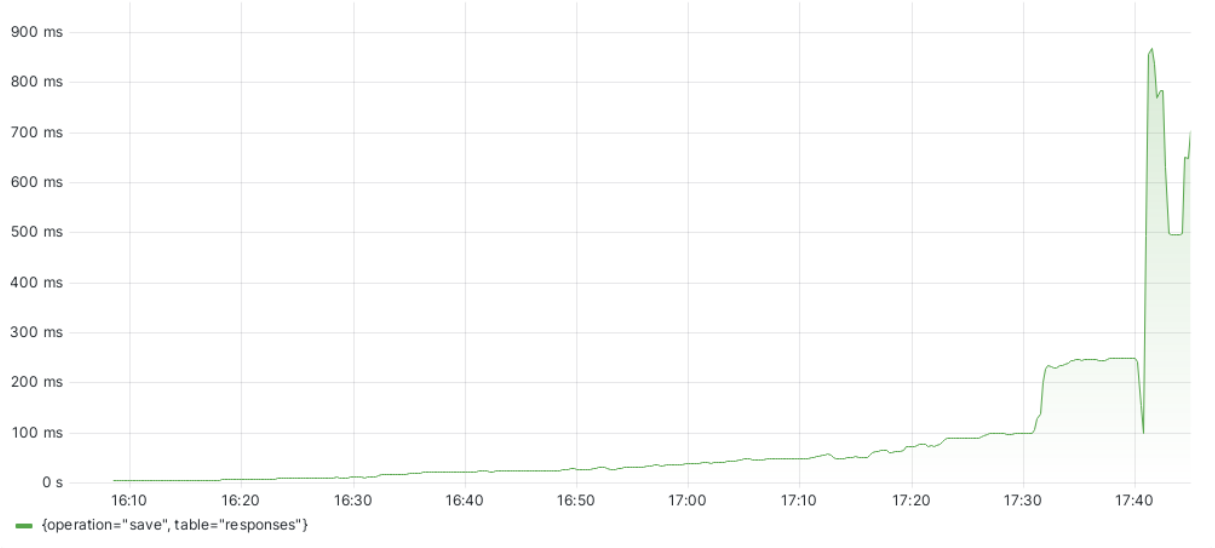


(b) Network Traffic – `cassandra-1`

Figure 4: Cassandra Load Test 1



(c) Pending Tasks – `cassandra-2`



(d) Crawler Operation Duration (P_{99})

Figure 4: Cassandra Load Test 1

Load Test 2

For the second load test, only a single Cassandra instance with 16 vCPUs and 32 GB of RAM was used, along with the same client specifications listed in table 8. In theory, if a singular, more powerful Cassandra instance was superior, it should reach or surpass the threshold of 400 concurrent workers per client instance, which was found in the first load test. Unfortunately, the worker's error rate sharply increased with only two concurrent workers per client instance.

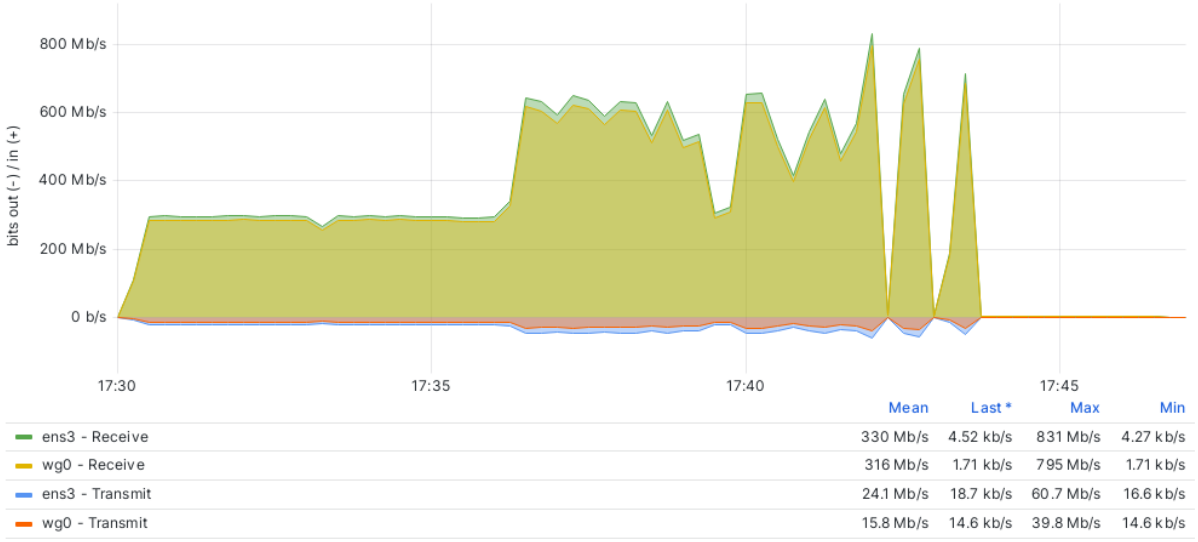
However, the Cassandra instance did not reach any computing or memory limits. The CPU usage maxed out at 62 %, while only 20 of the available 32 GB were used. During the test, network throughput on the Cassandra instance reached a maximum of 831 Mbit/s, which is visualized in fig. 5a. The net throughput rate within the Wireguard tunnel peaked at 795 Mbit/s.

Interestingly, the TCP error rate in fig. 5b correlates with sudden drops in network throughput that can be seen in the last quarter of fig. 5a.

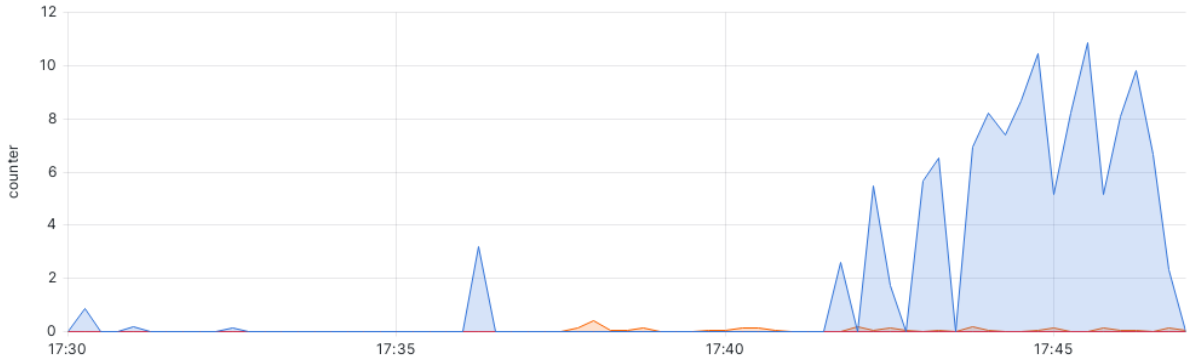
These results indicate that the network link, indeed, is a bottleneck. While the exact cause was not investigated any further, it is conceivable that either performance limitations of the Wireguard VPN tunnel or the larger number of concurrent connections may be responsible. Since the Cassandra cluster of three smaller instances achieved significant better performance, using this setup was preferred for the crawling process.

Name	Instances	Flavor	vCPUs	RAM
Cassandra	1	m1.xxlarge	16	32 GB
Dynamic Crawler	6	m1.large	4	8 GB
Static Crawler	7	m1.medium	2	4 GB

Table 8: Instance Specifications for Cassandra Load Test 2



(a) Cassandra Load Test 2 – Network Traffic



(b) Cassandra Load Test 2 – TCP Errors

Figure 5: Cassandra Load Test 2

5.4.2. RabbitMQ

For load-testing the queue component RabbitMQ, the same setup of crawler instances described in section 5.4.1 was used to simulate an operational scenario that is as realistic as possible. With that goal in mind, each client repeatedly published exemplary crawl job task entries to a single queue on the message broker. The entries had the same data structure used during the crawling process. The referenced domain names are randomly generated.

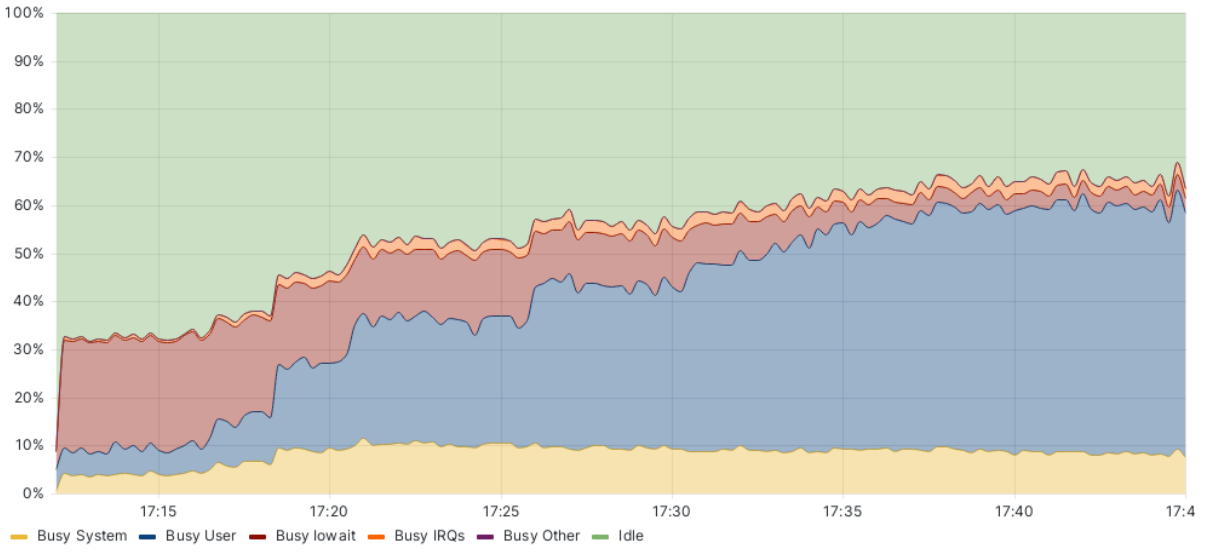
Load Test 1

Initially, the single RabbitMQ instance was provisioned with 4 vCPUs and 8 GB of RAM, as detailed in table 9. Similar to the Cassandra load tests, all clients gradually increased the number of concurrent worker *goroutines* that published tasks to the queue. The following steps in the number of concurrent workers per instance were synchronously used across all clients: 1, 2, 5, 10, 25, 50, 100, 200, 400, 800, 1,600, and 5,000.

Name	Instances	Flavor	vCPUs	RAM
RabbitMQ	1	m1.large	4	8 GB
Dynamic Crawler	6	m1.large	4	8 GB
Static Crawler	7	m1.medium	2	4 GB

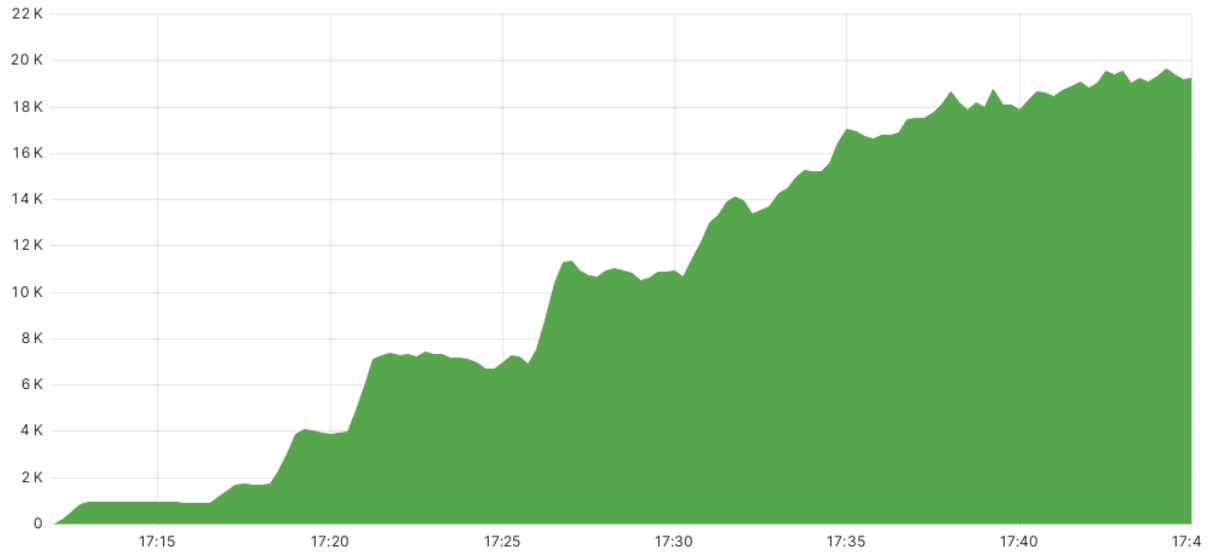
Table 9: Instance Specifications for RabbitMQ Load Test 1

Figure 6a shows an initial increase in CPU usage, but a plateau is reached around 65 %. At the same time, the number of published messages on the queue depicted in fig. 6b steadily increased. At 800 concurrent workers per client instance, fig. 6c shows the duration of queue publish operations beginning to increase. In the 99^{th} percentile, the duration per operation peaks at 5 s for 5,000 concurrent workers per client.

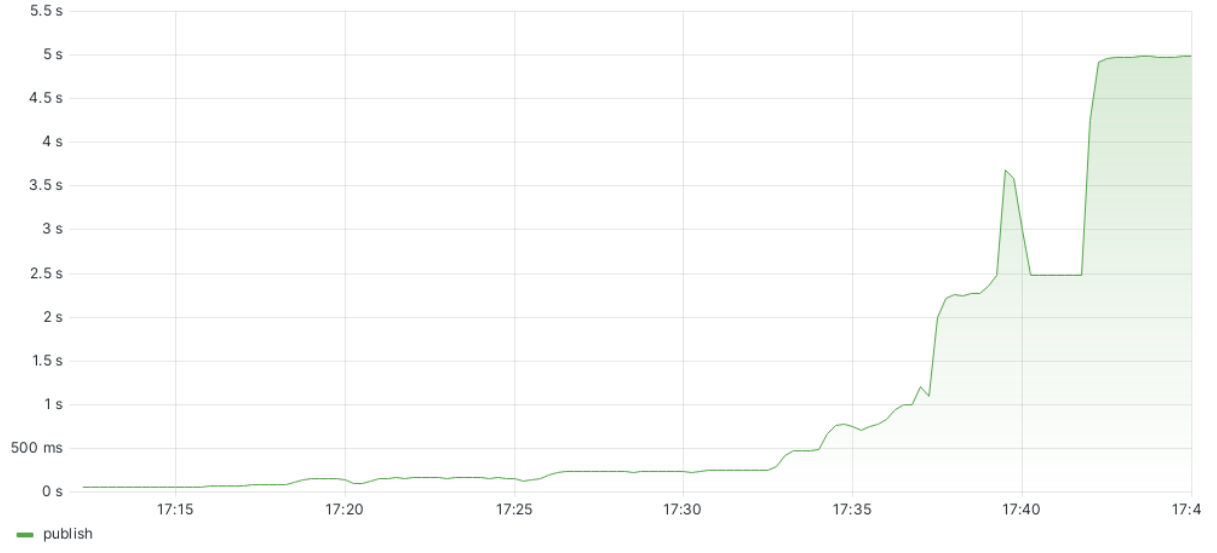


(a) CPU Usage

Figure 6: RabbitMQ Load Test 1



(b) Messages Published per Second



(c) Crawler Operation Duration (P_{99})

Figure 6: RabbitMQ Load Test 1

However, no errors occurred at a rate of 20,000 published messages per second. The first RabbitMQ load test revealed that utilizing a VM with fewer available resources was justifiable as the crawling process did not come close to this message rate.

Load Test 2

Table 10 lists the instances used in the second load test targeting RabbitMQ. This time, the message broker was downsized to 2 vCPUs and 4 GB of RAM. The same steps to increase concurrent workers per client instance were also in the second test. In an effort to provoke an error, after reaching 5,000 workers, the number was increased to 10,000 and 20,000 before the load test was concluded.

Name	Instances	Flavor	vCPUs	RAM
RabbitMQ	1	m1.medium	2	4 GB
Dynamic Crawler	6	m1.large	4	8 GB
Static Crawler	7	m1.medium	2	4 GB

Table 10: Instance Specifications for RabbitMQ Load Test 2

This time, CPU usage on the RabbitMQ instance plateaued at 85 %, while the allocated RAM peaked at 3.5 GB. The maximum network throughput was only 41.9 Mbit/s. While the increase to 20,000 concurrent workers per instance, which resulted in a total of 280,000 *goroutines*, drove up the client's publish operation duration to 10 s, still, no error occurred. With the smaller VM configuration, the number of published messages per second peaked at 16,000, which is still plenty for the intended use case. Thus, the specifications of the second load test were used to provision the RabbitMQ instance in the final crawling deployment.

5.5. Infrastructure Optimizations

This section describes the final steps before starting to crawl the web, as well as infrastructure optimizations applied during the crawl to increase performance.

5.5.1. Cassandra Optimizations

To make sure that the block storage used by Cassandra could be expanded if the need to do so arose due to potential off-base estimates or other issues, a final test was performed where each block storage volume size for the three Cassandra instances was reduced to 100 GiB. Then, load-testing clients were used to fill the volumes with HTML documents containing randomly generated content. After disk space was exhausted, all VMs were shut down, and the relevant Terraform configuration was modified to increase the provisioned block storage volume size to 980 GiB each. Next, the Cassandra VMs were rebooted, and Ansible was used to stop all Cassandra service daemons, expand the mounted filesystem on the block storage devices to the newly available size, and start the Cassandra daemons. After the Cassandra cluster was fully booted, Cassandra Query Language (CQL) queries were used to test whether the wide-column store worked as expected. As the disk expansion test was successful, it could be assumed that an expansion of storage capabilities would have been possible if it had become necessary.

Cassandra supports using several compression algorithms to reduce the disk footprint of stored data. Their documentation compares the available options and gives suggestions depending on the use case [13]. Because *Zstandard*, which is a relatively new development by *Facebook*'s parent company *Meta*, has a significantly better compression ratio than *LZ4*, Cassandra's documentation suggests using it over *LZ4* for storage-critical applications [13, 50]. However, because of its slightly faster compression and decompression speed, *LZ4* is recommended for performance-critical applications [13, 49]. Since section 3.1.1 roughly estimates the required storage capacity with a conservative compression factor of 1.5, the default *LZ4* ratio of 2.1 allowed for some headroom. To optimize for store and query operation speed, the Cassandra documentation's recommendation of using *LZ4* [13] was thus used to create all necessary tables in the CQL schema.

5.5.2. Crawler Optimizations

In an effort to optimize crawler performance and catch errors early, a slow ramp-up of crawling operations was carried out when starting the main crawl. Initially, the first thousand sites from the *Tranco* list selected in section 4.1.1 were dispatched on the static crawling queue. After that, a detailed analysis of the collected metrics and logs could be performed without risking that unusable data was being produced, which would have put an unnecessary load on the subsequent sites. Indeed, some errors arose that could be fixed while crawling was effectively paused. After deploying an updated crawler version to all instances, the first thousand websites had to be re-crawled. Because of the high number of requests that hit these most popular sites anyways, one can argue that a second visit within a short amount of time is ethically justifiable.

After analyzing the second set of results, no more errors were found and the first 100,000 sites – except for the already visited ones – from the *Tranco* list were dispatched on the static crawling queue. While monitoring the crawlers' health and throughput, the number of concurrent workers per crawler instance was slowly increased until the crawlers were on the verge of reaching VM limits, thereby optimizing their crawling performance. Table 11 shows the evolution of the crawler specifications throughout the crawl. Section 6 explains why iteration 7 in table 11 drastically reduced the number of static crawlers. After the first 100,000 sites, the remainder of the top one million sites from the *Tranco* list was dispatched in three additional batches.

Iteration	Static Crawlers			Dynamic Crawlers		
	Flavor	Instances	Workers	Flavor	Instances	Workers
# 1	m1.medium	6	20	m1.large	8	10
# 2	m1.medium	4	30	m1.large	9	12
# 3	m1.medium	4	30	m1.large	9	18
# 4	m1.small	5	30	m1.large	10	20
# 5	m1.small	5	30	m1.large	10	25
# 6	m1.small	5	30	m1.large	10	22
# 7	m1.small	1	30	m1.large	11	22

m1.small: 1 vCPU, 2 GB RAM · m1.medium: 2 vCPUs, 4 GB RAM · m1.large: 4 vCPUs, 8 GB RAM

Table 11: Evolution of Crawler Specifications Throughout the Crawl

While six static crawlers had been initially provisioned with 20 concurrent workers per instance, it was possible to increase the number of concurrent workers per instance to 30 and reducing the VM flavor from *m1.medium* with 2 vCPUs and 4 GB of RAM to *m1.small* with 1 vCPUs and 2 GB of RAM while maintaining a good task rate.

At the same time, it was possible to shift some resources from static to dynamic crawlers by shutting down static crawler instances and creating additional dynamic crawler instances thanks to the automated, idempotent provisioning and deployment using Terraform and Ansible. This shift was made after re-evaluating the task consumption rate of both crawler types. The browser-automated dynamic crawlers have a lower throughput than the static crawlers, so providing additional resources to dynamic crawling made sense.

However, job throughput alignment was still suboptimal with the specifications of iteration 4 in table 11. The reason why the static crawler instance count was not reduced further was to avoid concentrating too much HTTP traffic on one instance and one IP address. Using multiple outgoing IP addresses may help to reduce the chance of getting blocked by organizations seeing traffic on multiple site, like the major CDN providers. Hence, it was necessary to balance the job throughput rate with the amount of individual crawling hosts and their IP addresses.

6. Results

After sections 3 to 5 described the crawler's architecture, implementation, and deployment, this section details the course of the crawling process and analyzes the resulting content corpus.

Unfortunately, the intended crawl volume of Tranco's one million most popular websites could not be obtained because of a dispute with *bwCloud* that ultimately led to a ban on crawling activity and halted VMs on very short notice. It is important to note that the necessary application to request cloud resources from *bwCloud* for this thesis had been made correctly, including explicitly requesting resources for web crawling, as their terms of service demanded. Regrettably, personnel at *bwCloud* stated to have read over this explicit request and refused to take responsibility by sticking with the original project approval.

Due to this circumstance, only the first 624,780 sites on the Tranco list[26] [117] could be crawled for subsequent analysis. Nonetheless, the collected sample is large enough to provide relevant conclusions in regards to R1 and R2.

6.1. Infrastructural Analysis

In addition to analyzing web content, it is worth considering the infrastructure enabling this data collection. This section first evaluates the distribution of IP version support among crawled sites before evaluating how the provisioned crawling infrastructure held up during the process.

6.1.1. IPv6 Adoption Rate

Because of the peculiarities of the *bwCloud* environment described in section 5.3, special attention to the connection paths for the individual IP versions was required. Hence, metrics on the adoption of IPv6 were collected during the crawl.

Of all targeted 624,660 sites, only 23.1 % supported dual-stack connectivity, i.e., serving both `A` and `AAAA` records for their domain. The most significant fraction, 65.1 %, still only supported IPv4 by merely serving `A` records. 0.07 % of sites are only reachable through IPv6, while 11.7 % did not serve valid `A` and `AAAA` records. Many of the latter sites presumably only serve subdomains, for example, because they are part of a CDN like `l-msedge.net` (Microsoft Azure) or `googleusercontent.com` (Google). Note that queries were only performed for the root domain without considering any subdomains. For instance, a DNS query for `A` and `AAAA` records was performed for `example.com`, while `www.example.com` was not evaluated.

Unfortunately, these results paint a bleak picture of IPv6 adoption among the most popular sites on the web, even though the technology was introduced over twenty years ago. One may also assume that the adoption rate is even lower for less popular sites.

6.1.2. System Load

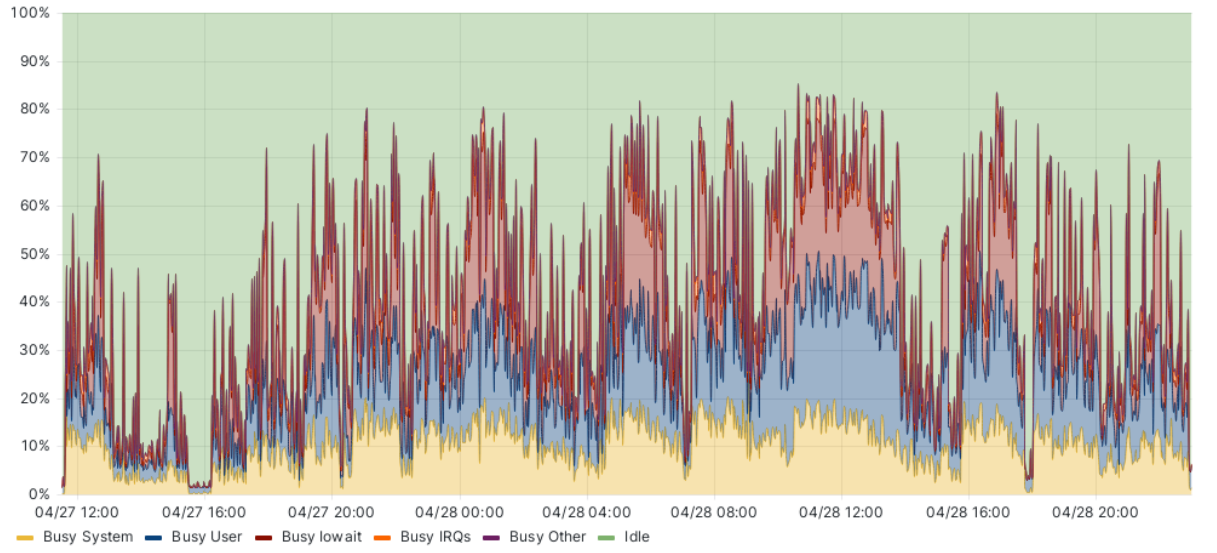
This section analyses some of the metrics collected during the crawl and evaluates whether the estimations from sections 3.1.1 and 5.4 were correct. It also describes some of the issues discovered during and after the crawling process, which are further discussed in section 7.2.1.

[26] Available at `https://tranco-list.eu/list/3V6KL`

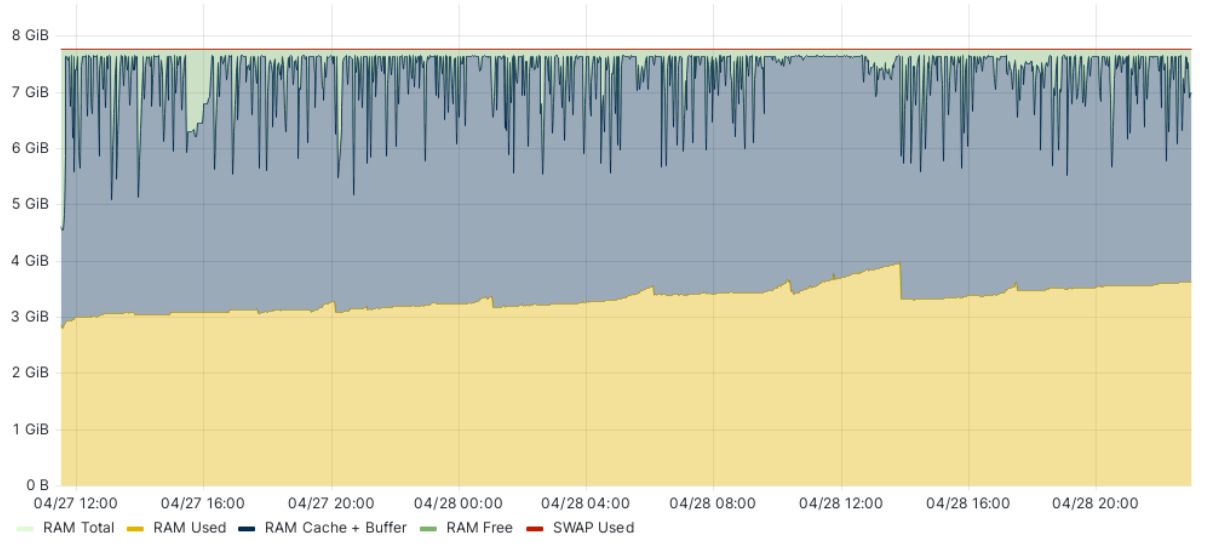
Cassandra

All three Cassandra instances showed roughly the same load and behavior during the crawl. Figure 7 visualizes the collected metrics of the first Cassandra instance over the main crawl time period. The other instances' visualized metrics look very similar. Figure 7a shows that the CPU was utilized in a healthy manner. While not many machine resources were idling, they were not exhausted, either. Half of the available RAM in fig. 7b was filled with data, while the remainder was used for caching and buffering.

Similar to the load tests in section 5.4.1, the network throughput in fig. 7c peaks at 353 Mbit/s. For the other instances, the respective maximum is 309 Mbit/s and 350 Mbit/s. From a simplified mathematical point of view, the network link of a single Cassandra instance would indeed have been a bottleneck in the deployment with the sum exceeding the 1 Gbit/s limit.

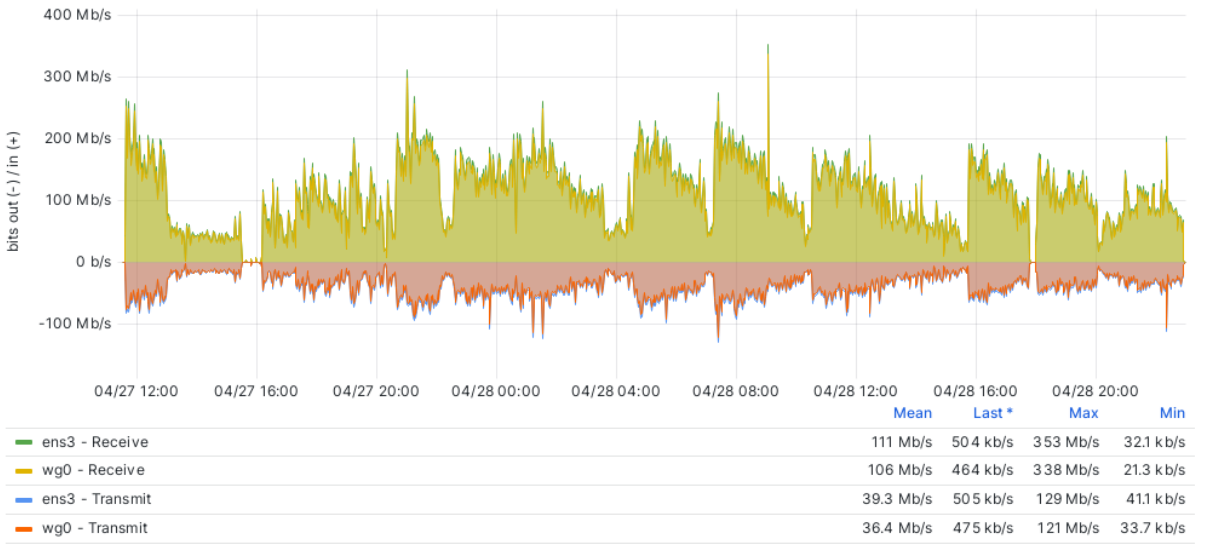


(a) CPU Usage

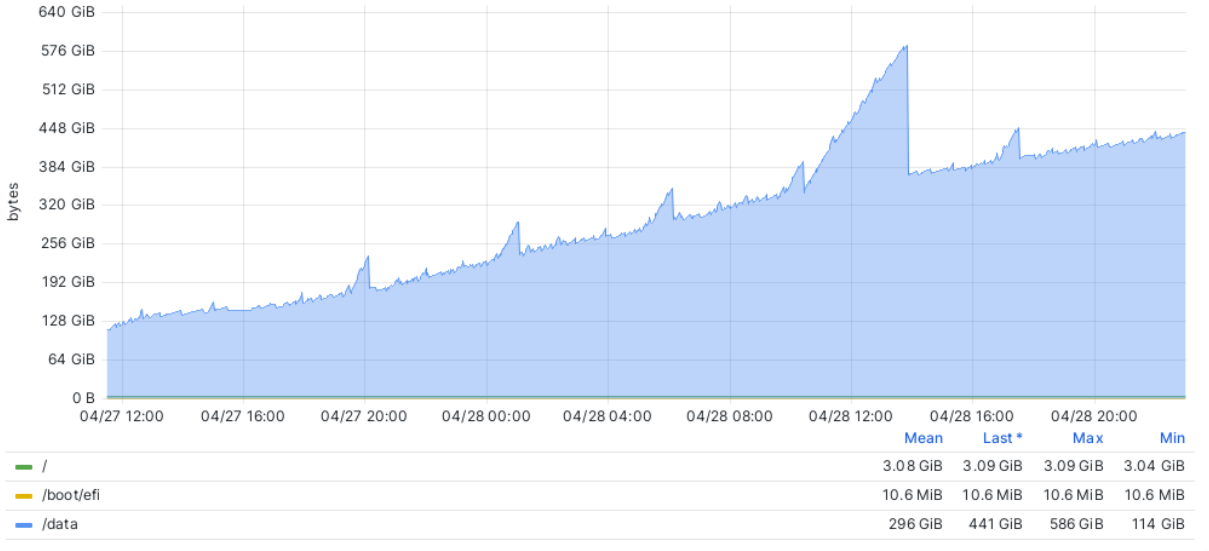


(b) Memory Usage

Figure 7: Node Metrics – `cassandra-1`



(c) Network Traffic



(d) Disk Usage

Figure 7: Node Metrics – `cassandra-1`

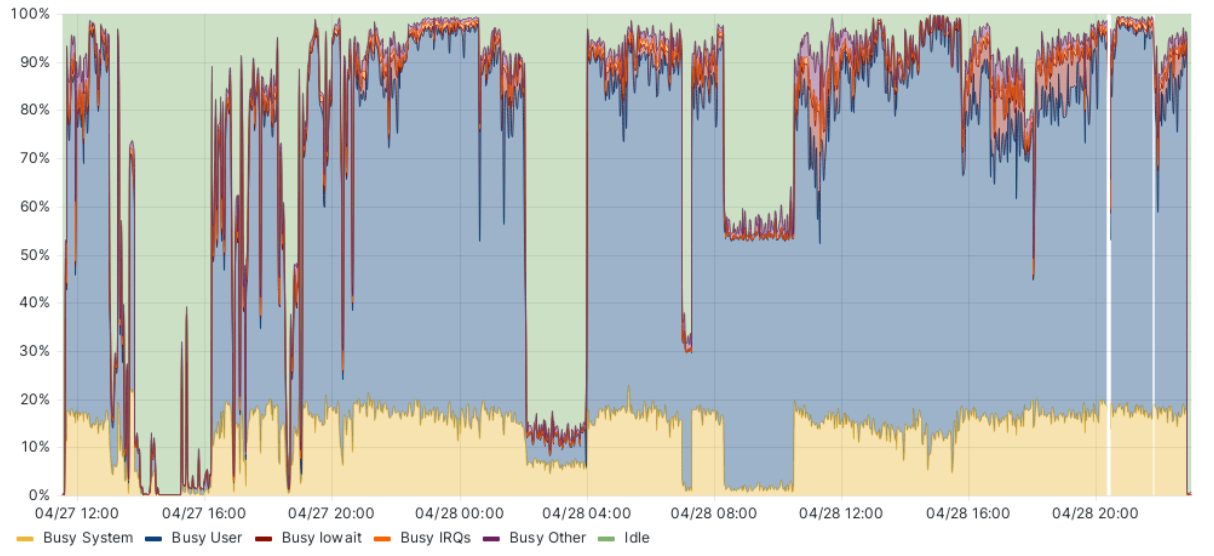
A rather interesting disk usage behavior emerges in fig. 7d. Roughly speaking, the amount of stored data grew linearly. However, a steep increase followed by a sharp decrease can be identified every four to five hours. This behavior is shared among all Cassandra instances. In one case, the data grew at an increased rate for several hours before being drastically reduced to the expected trend in data growth. While it is unclear what caused these disk usage spikes, it may be related to a scheduled data compaction or clean-up process.

Dynamic Crawlers

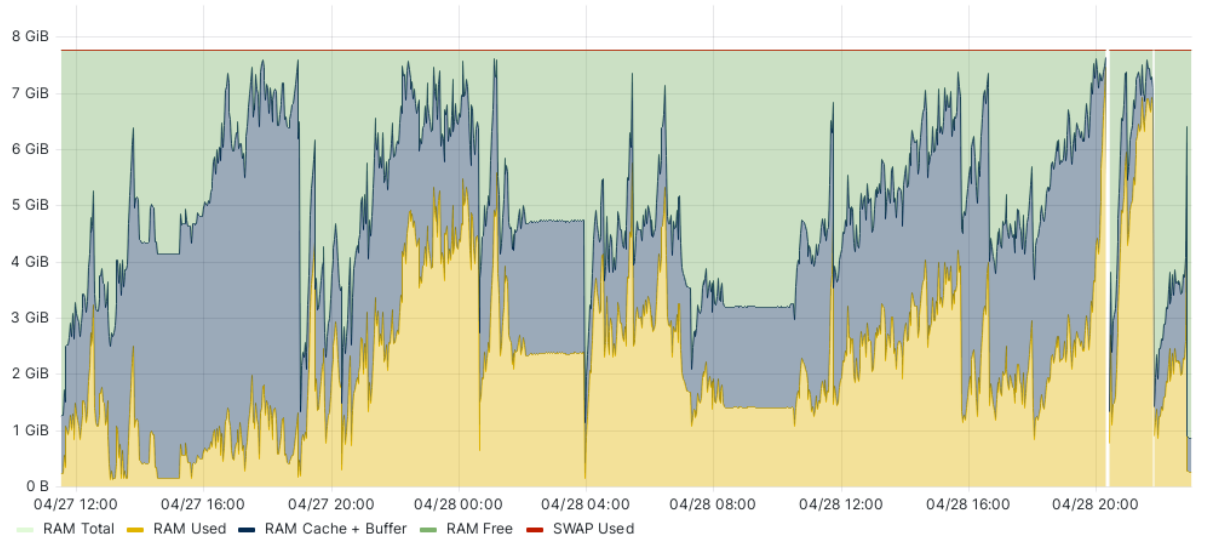
The dynamic crawler instances were provisioned to optimize crawling performance by running as many as possible concurrent worker *goroutines* and, in extension, Chrome tabs within the

instance's shared Chrome window. Because of the complex nature of remote-controlling a web browser to visit sites with unknown behavior over a prolonged period, several issues occurred where workers died, or the instance's disk space was exhausted because of bugs in the library interacting with Chrome. Section 7.2.2 discusses these issues and necessary countermeasures in detail.

Figures 8a and 8b show that, during certain periods, CPU and RAM resources on one of the dynamic crawler instances were used to their limits. What must be noted is that system load varies depending on the effort needed to render a particular visited page. In other time ranges, for example, on April 28^{th} between 08:15 and 10:30, performance degradation occurred due to instance-bound persistent failures. The other dynamic crawler instances show similar patterns in their resource usage.

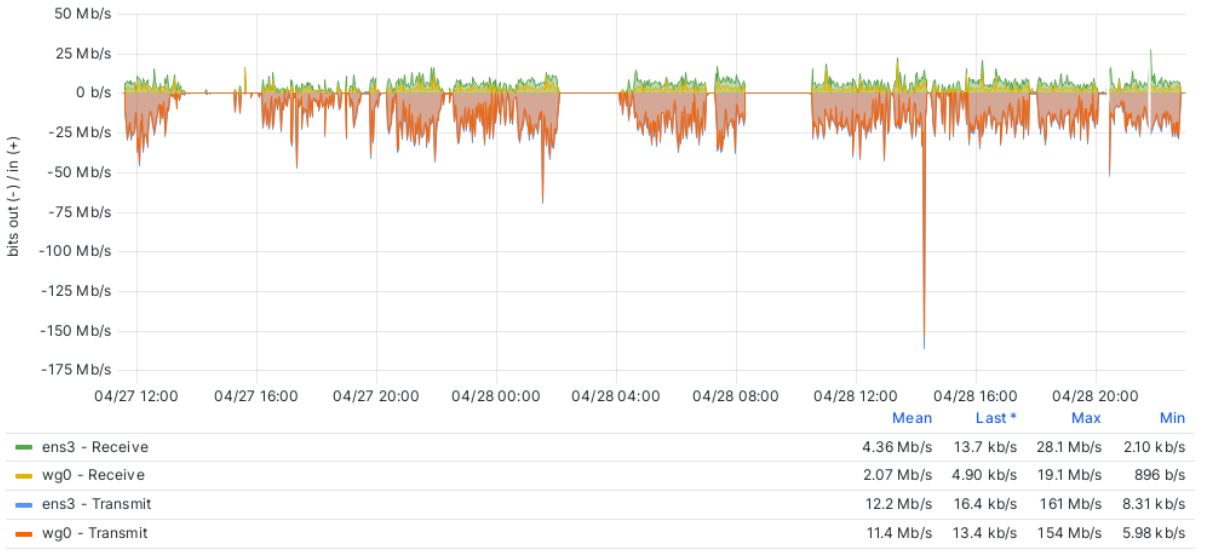


(a) CPU Usage

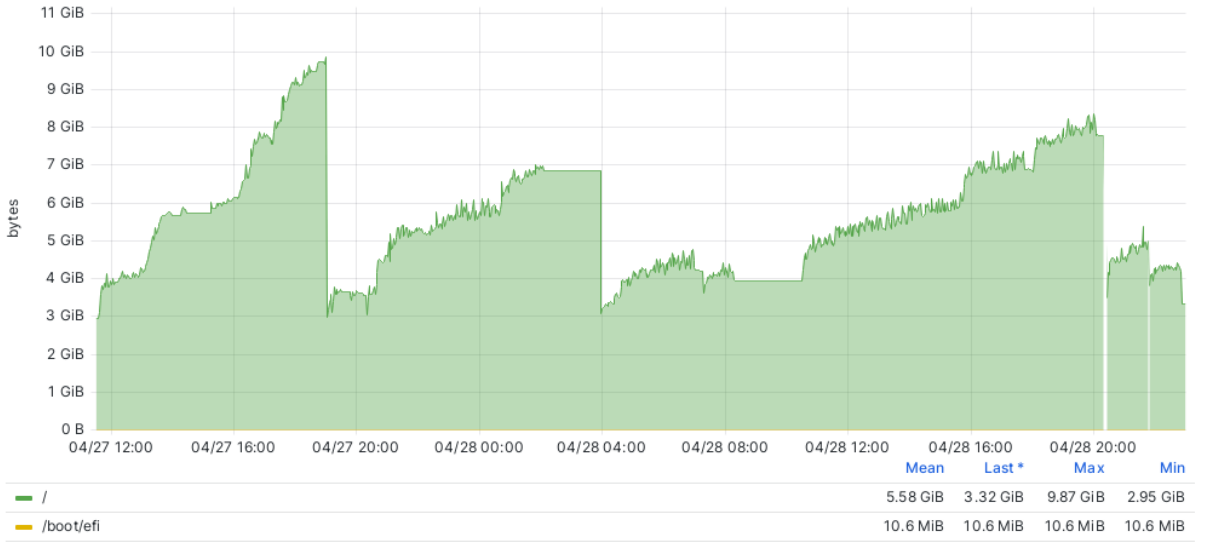


(b) Memory Usage

Figure 8: Node Metrics – `crawler-dynamic-1`



(c) Network Traffic



(d) Disk Usage

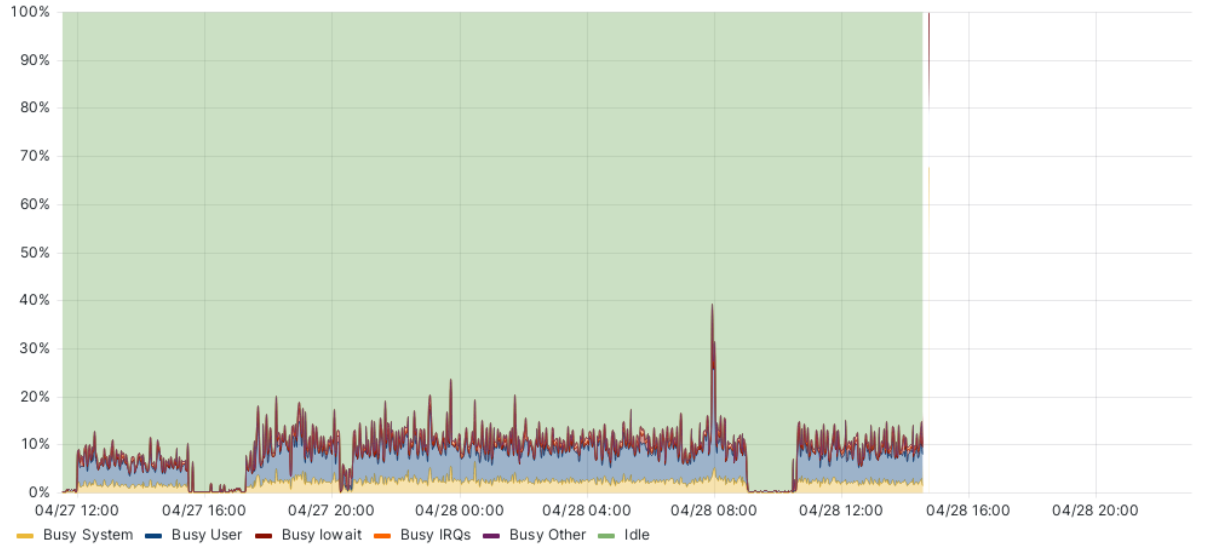
Figure 8: Node Metrics – `crawler-dynamic-1`

The individual network traffic of the dynamic crawlers was manageable, with the exemplary instance in fig. 8c having a maximum transmitting data rate of 161 Mbit/s (tunneled traffic plus VPN overhead), while the maximum receiving data rate was just 4.36 Mbit/s. As these VMs were mainly limited by their compute and memory resources, networking was not an issue.

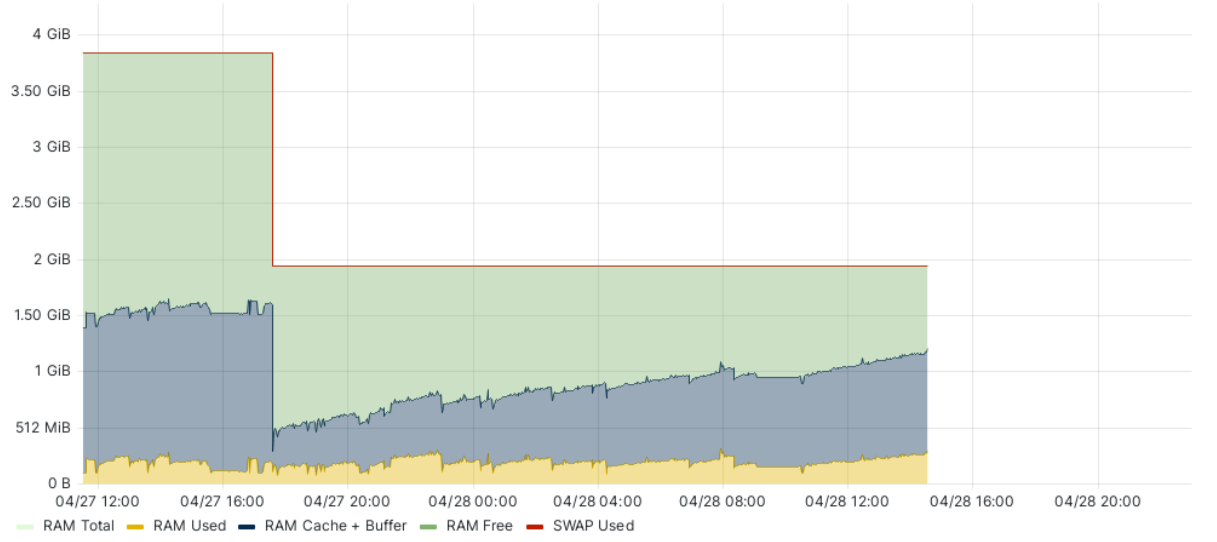
The previously mentioned crawler disk hogging is visible in fig. 8d. Because temporary data from Chrome and *chromedp* was not cleaned up correctly, a scheduled shutdown of workers was necessary to remove any dangling data filling up the instance's disk. These drastic data reductions can be clearly seen in fig. 8d.

Static Crawlers

Fortunately, the less-complex static crawlers were significantly more reliable. As previously addressed in section 5.5.2, the available VM resources per instance were not used to their full potential because the load distribution over more IP addresses was considered more important. Figure 9 shows the resource usage for a sample instance. Even when reducing the available resources by downgrading their VM flavor to *m1.small*, as described in section 5.5.2, resource usage remained quite low. Figure 9b nicely visualizes the change in available RAM. Since the dynamic crawlers were significantly slower and the assigned quotas prevented increasing the total amount of instances, it would have made little sense to increase the number of concurrent workers per static crawler instance, even though the available resources were very sparingly used.



(a) CPU Usage



(b) Memory Usage

Figure 9: Node Metrics – `crawler-static-1`

NAT64 Gateway

Due to *bwCloud*'s IPv6-only connectivity described in section 5.3, a NAT64 gateway was deployed in another region that could route packets to IPv4-only web servers. The gateway's computing and memory resources were generously specced with 4 vCPUs and 8 GB RAM (*m1.large*) to prevent it from becoming a bottleneck in the crawling process.

Throughout the crawl, the NAT64 gateway VM used a maximum of 10 % CPU and 3.5 GB RAM. Figure 10 shows the network throughput peaked at only 139 Mbit/s. It is thus reasonable to assume that the NAT64 workaround did not negatively affect crawler performance in a significant way.

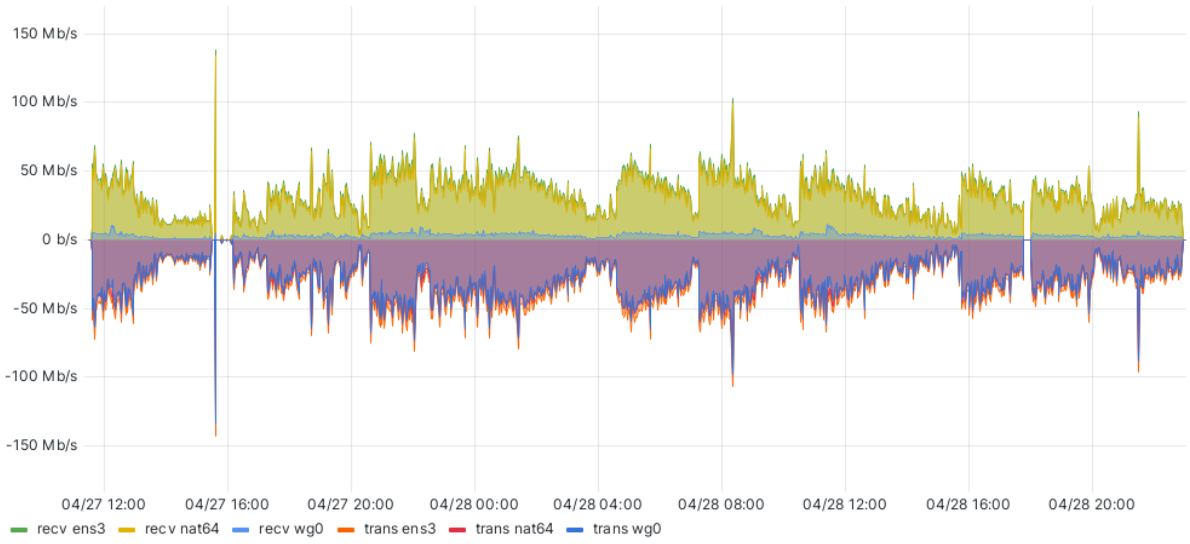


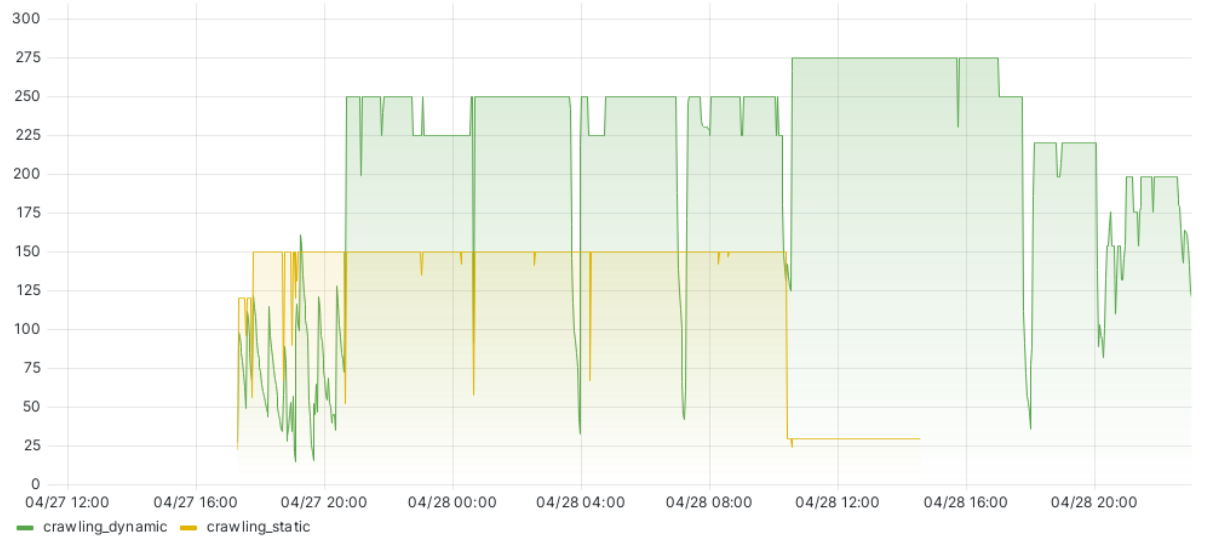
Figure 10: Network Usage – `nat64-gateway`

Crawl Statistics

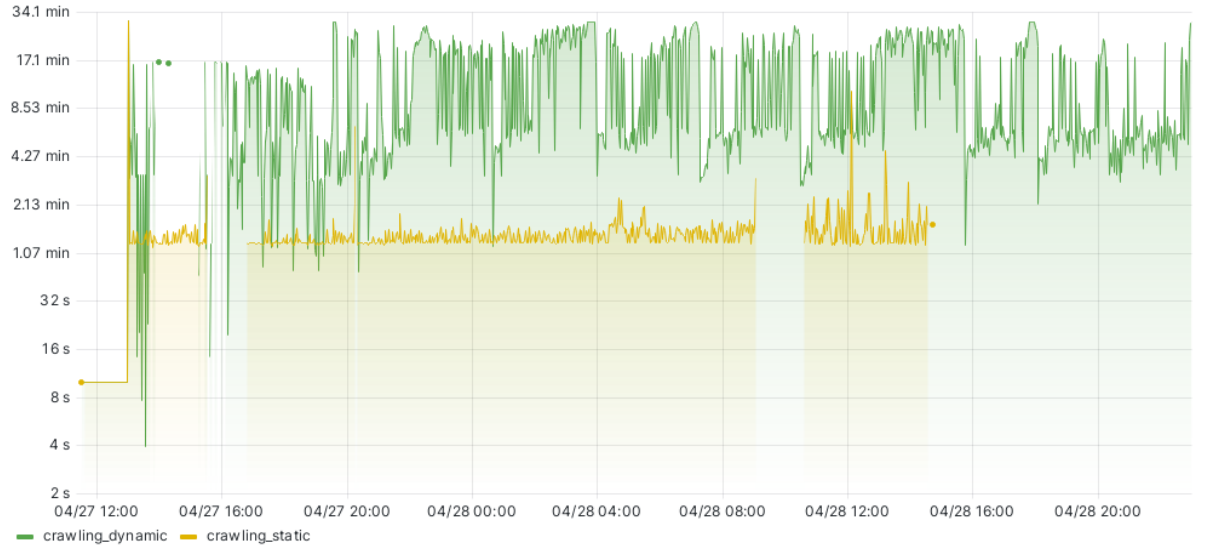
Several custom metrics detailed in section 5.2 have been collected during the crawling process. Figure 11a visualizes the number of active concurrent workers per job queue. Some initial errors on the dynamic crawlers caused the number of their workers to fluctuate at the beginning of the crawl. After that, the total number of concurrent dynamic workers was $\approx 40\,\%$ higher than the number of static workers to compensate for the slower job throughput rate. The previously mentioned scheduled cleanup of growing temporary data, which required to shutdown all workers, is also visible in fig. 11a. The sudden drop of static workers on April 28^{th}, at around 10:30, can be explained with *bwCloud*'s forced shutdown of static crawler VMs. While attempts were made to resolve the issue with *bwCloud* personnel, the remaining jobs in the dynamic crawling queue had been continued to be processed to allow for full coverage of all sites that were already visited by the static crawlers.

Figure 11b shows the job duration in the 99^{th} percentile on a logarithmic scale. While in extreme cases, static crawlers relatively consistently took 1.6 min to complete a crawling task, i.e., visiting a targeted domain's home page and any subsequently discovered authentication-related URLs, the dynamic crawlers' task duration exhibited a significant deviation. In the 99^{th} percentile, values jump between around 5 min and 15 to 30 min. That is interesting because the dynamic crawler implements a fixed timeout of 20 min for every queue task. After that,

the handler's *context* is canceled, and the current blocking operation is supposed to return an error. As some tasks appear to have been running longer than that, there may be a bug in the crawler implementation or in the `chromedp` library. Unfortunately, this behavior was only discovered after the crawl had already concluded, so no further investigation was possible. Nonetheless, fig. 11b shows the drastic difference in crawl time per site that makes crawling client-side rendered content significantly more challenging with limited resources.

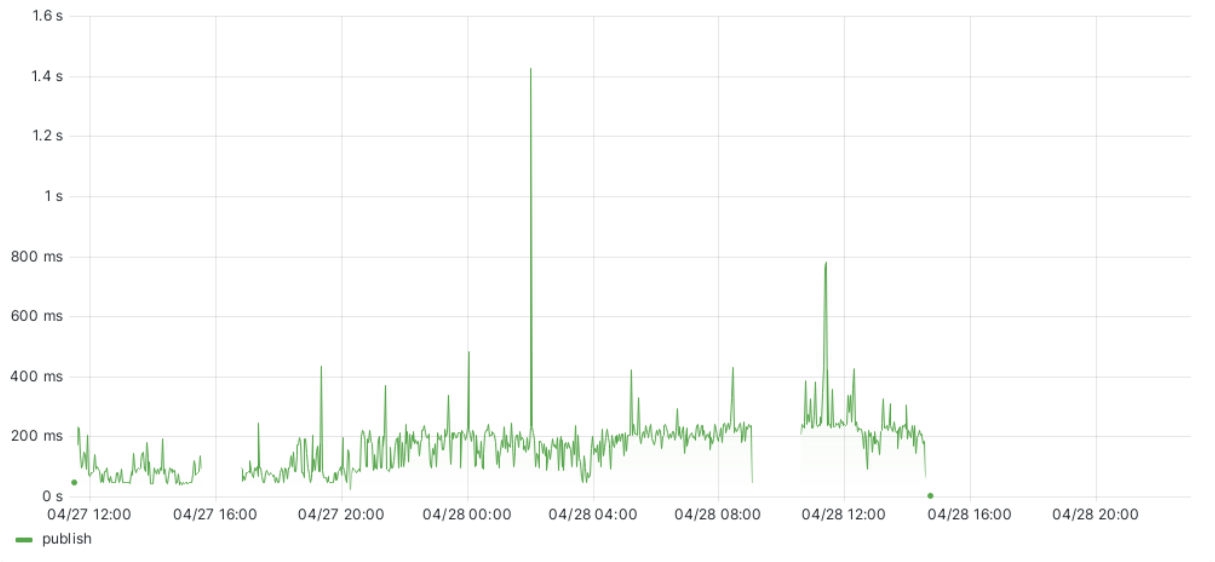


(a) Active Workers per Queue

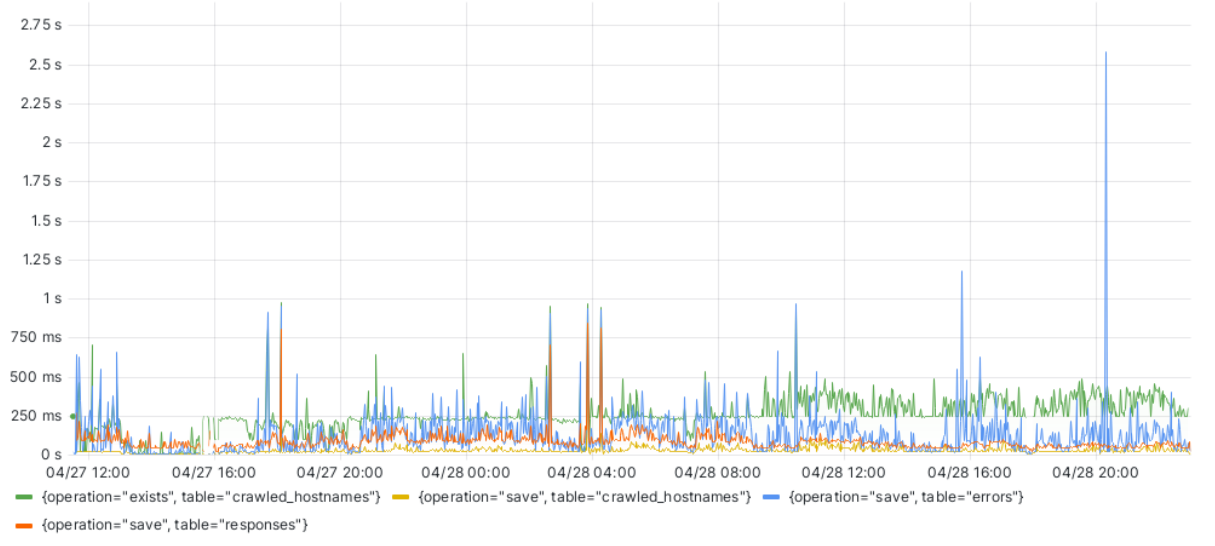


(b) Task Duration (P_{99}, log scale)

Figure 11: Crawler Load Metrics



(c) Queue Operation Duration (P_{99})

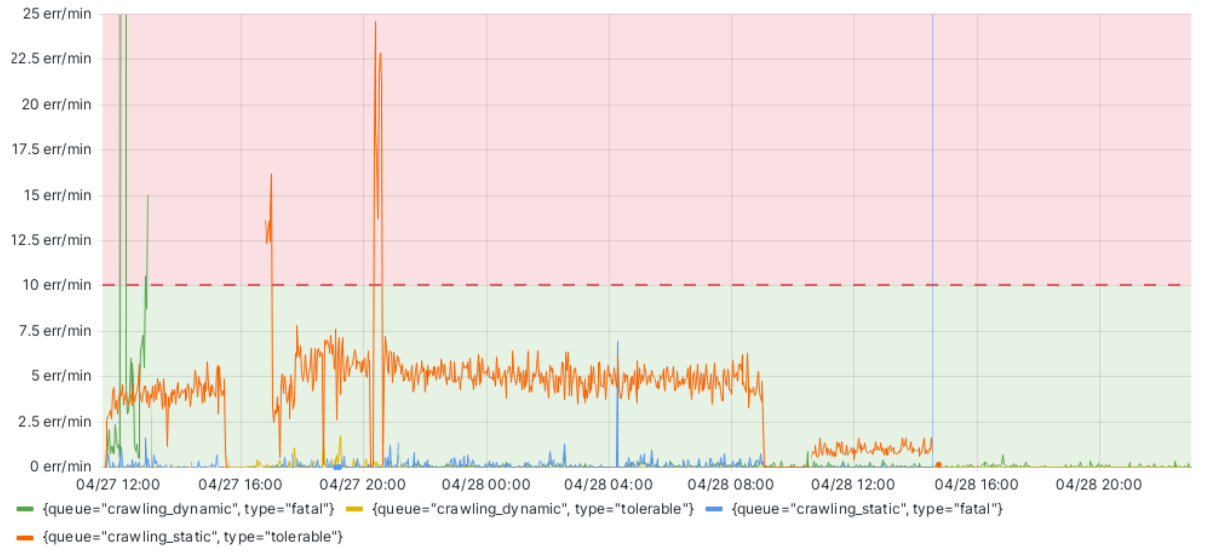


(d) Store Operation Duration (P_{99})

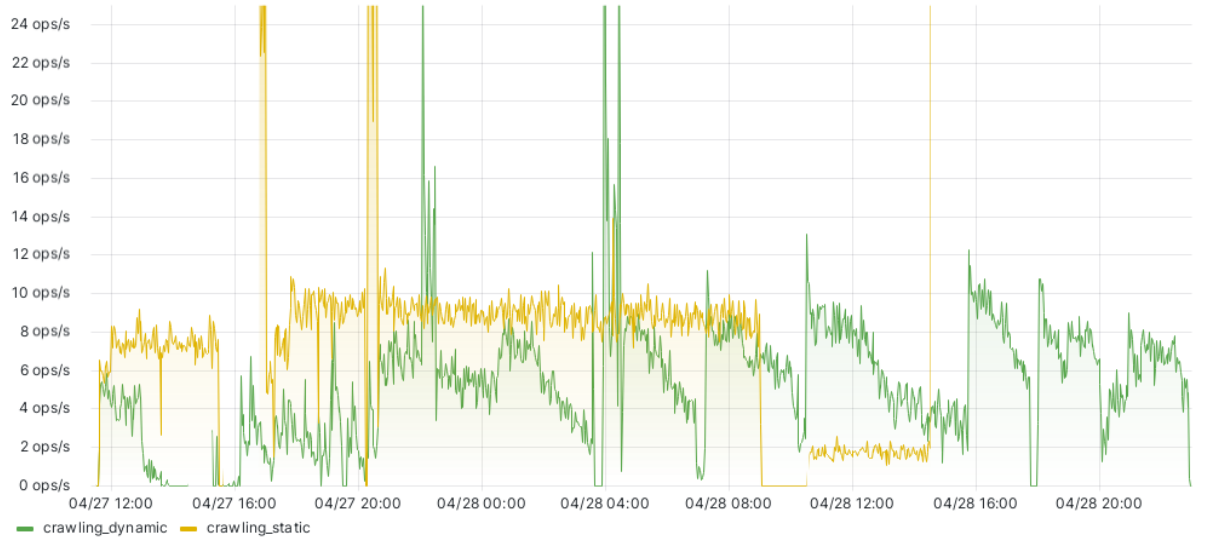
Figure 11: Crawler Load Metrics

In regard to queue publishing operations, RabbitMQ was easily able to handle the resulting load. Figure 11c shows that publish operations took around 200 ms in the 99^{th} percentile. This correlates with metrics collected for RabbitMQ's host machine, which place CPU usage in a range of $5 - 55\,\%$, on average at $\approx 10\,\%$. At the same time, RAM usage was just over $50\,\%$.

While read operations in the *crawled hostnames* table tended to take a bit longer than others, store operations overall had equally good performance, as displayed in fig. 11d. In the 99^{th} percentile, CQL queries mostly took 100 to 500 ms – with the exception of few outliers. That is significant because Cassandra had to ingest all fetched content sent by the crawlers. It appears that write performance was satisfactory for the use case. Section 7.2.5 discusses why that is only partly correct and how read performance compares.



(e) Total Error Rate



(f) Total Job Throughput Rate

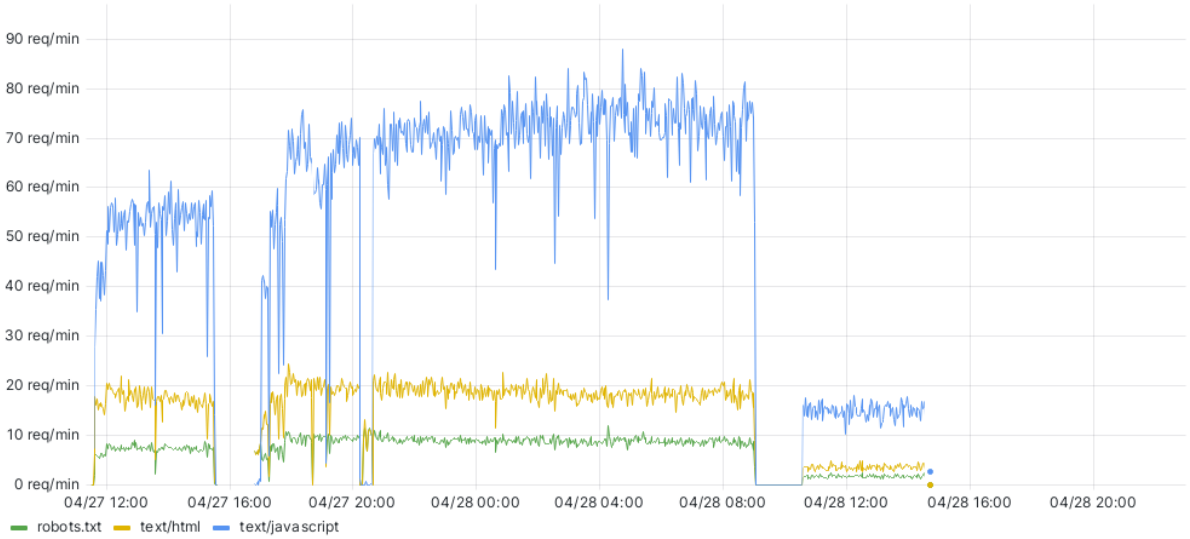
Figure 11: Crawler Load Metrics

In fig. 11e, the error rates of all crawlers by gravity and affected queue are visualized. The dashed red line represents an alerting threshold in Grafana, over which notifications were sent to request manual intervention. While the curve for the initial dynamic crawler failures, which were discussed previously, extends past the plotted area, most subsequent error rates were within an acceptable range.

The previously described issue of dying worker *goroutines* is also evident in fig. 11f, which shows the job throughput rate separated by crawler type. While the static crawlers processed jobs at a consistent cumulative rate of around nine tasks per second, the dynamic crawlers repeatedly started out with a similarly high throughput rate and then gradually lost momentum. After the scheduled clean-up operations, the dynamic crawlers processed between 6 and 8 tasks per second in total, before losing an increasing amount of concurrent workers, resulting in a rate of

only 2 tasks per second. The scheduled clean-up of temporary data stops the remaining workers and recreates the initially provisioned amount of concurrent *goroutines*, causing a steep increase in throughput.

Finally, fig. 11g shows the total request rate of all static crawlers by content type. Because JS resources are often separated into multiple files, which may even be served from different web servers or CDNs, the amount of issued JS requests per minute is almost four times the amount of HTML documents.



(g) Total Request Rate (Static Crawlers)

Figure 11: Crawler Load Metrics

6.2. Quantitative Analysis

This section reviews the collected data quantitatively and aims to partly answer the posed research questions R1 and R2 through statistical analysis.

6.2.1. Successful Connection Rate

Of all 624,780 targeted sites, $73.69 \pm 1.32\,\%$ served at least one response that could be fetched, while no successful connection was possible for the remaining $26.31 \pm 1.32\,\%$. As the standard deviation indicates, the differences in successful connection rate between static and dynamic crawlers are negligible.

On the initial home page visit, the static crawlers encountered 56,985 TLS-related errors, for instance because the certificate was expired or not valid for the requested hostname. 49,599 hostnames could not be resolved, presumably because only their subdomains are used to serve content, like `l-msedge.net` (Microsoft Azure) or `googleusercontent.com` (Google). Moreover, a timeout on the initial HTTP request occurred in 35,879 cases, while 19,016 hosts either refused, reset, or closed the connection. The dynamic crawlers recorded similar amounts of errors per category, although the individual counts vary slightly.

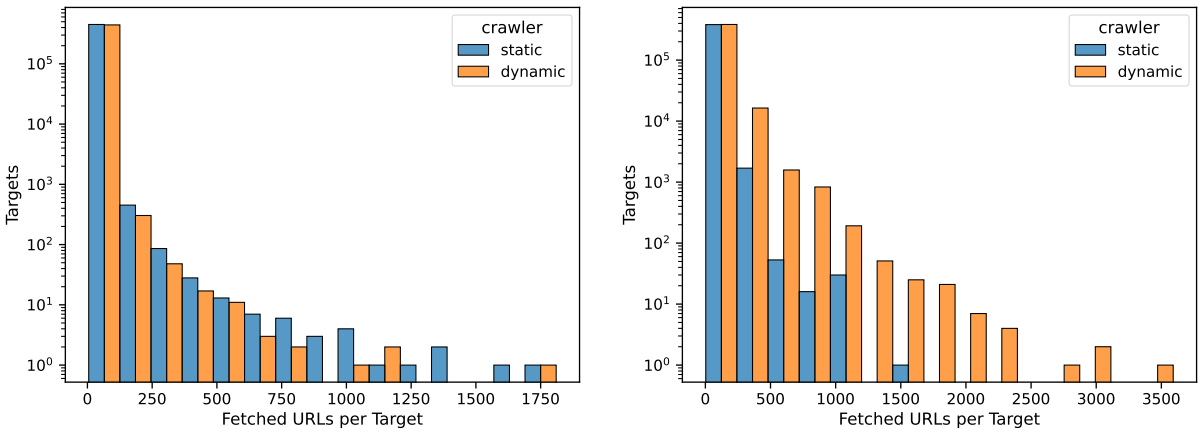
Of all captured HTTP responses, the majority had successful HTTP status codes. $98.93 \pm 0.36\,\%$ had a `200` *OK* status code, while only $0.42 \pm 0.25\,\%$ had a `403` *Forbidden* status code and

$0.26 \pm 0.12\,\%$ had a `404` *Not Found* status code. All other status codes combined appeared in $0.39 \pm 0.02\,\%$ of HTTP responses.

6.2.2. Discovered Content Distribution

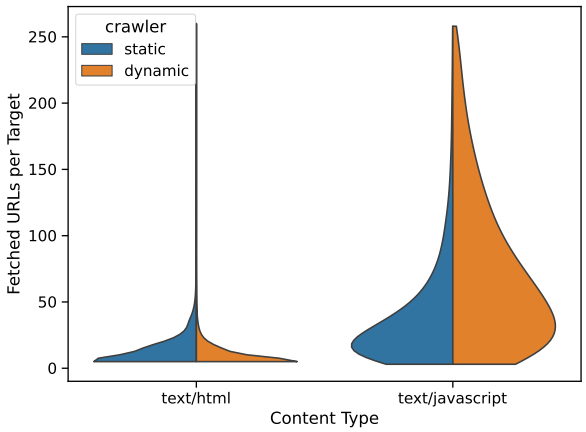
Figure 12 visualizes the distribution of previously discovered and successfully crawled content. Figure 12a shows the number of fetched HTML documents per targeted website separated by crawler type. The histogram's y-axis uses a logarithmic scale to illustrate the exponentially decreasing number of documents per site. The mean number of HTML documents per site collected by static crawlers is 12.87 ± 15.34. Such a high standard deviation implies that the amount per site varies considerably. Indeed, the percentiles $P_{10} = 5$, $P_{50} = 10$, $P_{90} = 25$, and $P_{99} = 55$ indicate high variance and the existence of outliers, which is visible in fig. 12a. For instance, 1,810 HTML documents were collected from *dnscentral.com* . This particular outlier is discussed in section 6.4.

What's also visible in fig. 12a is that the static crawlers actually discovered and crawled slightly more authentication-related URLs per site than the dynamic crawlers, whose mean value of HTML documents per target is 9.60 ± 11.44, with $P_{10} = 5$, $P_{50} = 5$, $P_{90} = 20$, and $P_{99} = 45$. Although the difference is not substantial, the exact cause is unclear.



(a) HTML

(b) JS



(c) Combined, $< P_{99}$

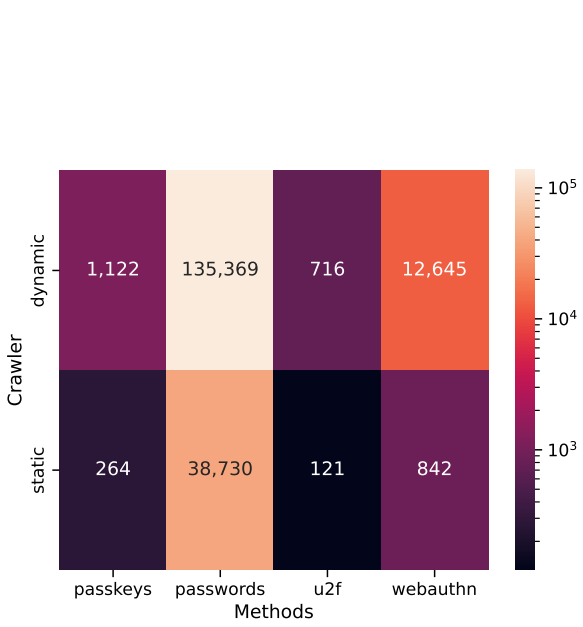
Figure 12: Distribution of Content Per Target

In contrast, fig. 12b shows the number of JS files crawled per website. It is apparent that the dynamic crawlers were able to discover significantly more content than the static crawlers with respective mean values of 84.45 ± 92.06 and 37.89 ± 41.08, while the variance is even greater. The percentiles of dynamically crawled JS per site are $P_{10} = 12$, $P_{50} = 60$, $P_{90} = 180$, and $P_{99} = 408$. On the other hand, the static crawlers discovered considerably less JS content, with $P_{10} = 6$, $P_{50} = 27$, $P_{90} = 81$, and $P_{99} = 195$.

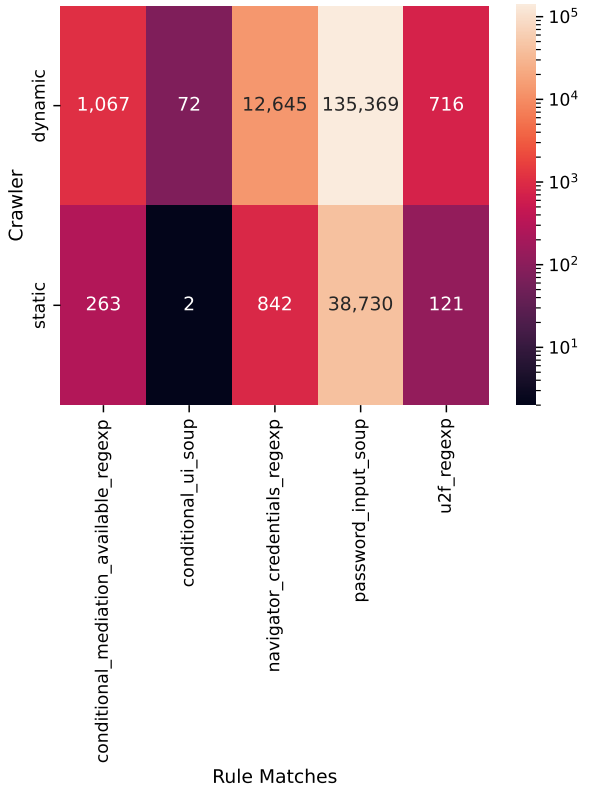
Figure 12c visualizes the amount of crawled content per site, separated by content type and crawler type in combination. To allow for better visibility, the data in fig. 12c is limited to $< P_{99}$ to prevent extreme outliers from compressing the common values on a linear scale. The graph nicely shows that significantly more unique JS resource URLs have been fetched per site compared to HTML documents. Additionally, the slightly higher number of HTML pages crawled statically is contrasted by the significantly larger quantity in dynamically crawled JS resources.

6.2.3. Authentication Method Detection

As outlined in section 4.3.2, matching was implemented using several Regex and DOM-based rues to detect the authentication methods used by websites. All crawled content was analyzed using this rule set to automatically detect support for the different studied authentication technologies. The same rule set was used for both types of crawler to compare the crawling technique.



(a) Detected Methods



(b) Matched Rules

Figure 13: Detected Authentication Methods per Site, Separated by Crawler Type

Figure 13 contains two heat maps depicting the matches found by either crawler type. While

fig. 13a shows the number of detected authentication methods separated by crawler type, fig. 13b represents the individually matched rules per crawler type. A combination of multiple rules may detect an authentication method. In both cases, duplicate matches per site are ignored, so only one rule match per target is considered.

It is apparent that content collected by the dynamic crawlers has a significantly higher rule-matching rate with a mean value of 12.9 ± 13.7. The highest difference occurs in the rule testing for *Conditional UI* support. While static crawlers could only identify 2 sites with *Conditional UI*, dynamic crawlers were able to identify 72 sites, which is *36* times more. The rule testing for a traditional password-based sign-in form had the smallest factorial difference of 3.5, although the absolute difference with static crawlers identifying 96,639 fewer sites than dynamic crawlers is still substantial.

As one would expect, the largest number of sites that had at least one match employ traditional password-based sign-in forms, with a total of 137,424 identified websites across both crawler types. In total, 12,690 sites were found to reference the WebAuthn *Credentials* API within their JS resources. It is important to note that this does not necessarily mean that every user has access to FIDO-based authentication on these websites, or that the relevant part of the contained code is even in use. The rules detecting WebAuthn *Credentials* API usage also do not allow any conclusions on whether FIDO credentials are being used in a single-factor or multi-factor authentication scheme.

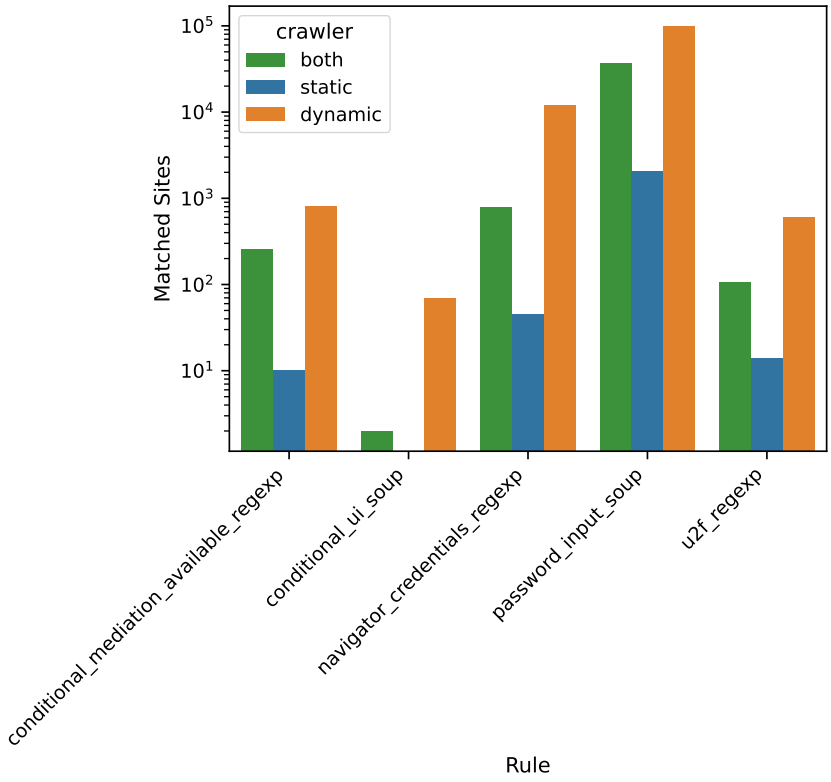


Figure 14: Matched Sites Per Rule and Crawler

In contrast, the rule testing for use of the `isConditionalMediationAvailable` method on the *Credentials* API identified a total of 1,077 sites, indicating passkey support. The same is true for the *Conditional UI* rule, which matched a total of 72 sites. Lastly, a total of 730 sites still

having references to the legacy *U2F* API in their JS resources were identified. Aside from the previously mentioned matches, the remaining 460,396 sites could not be matched by any of the employed rules for any of the two crawler types.

Figure 14 displays how many sites were matched by each rule for one or both crawler types. It is striking that while there are sites matched by both crawlers for every rule, the dynamic crawler was able to identify a larger amount of sites in every case. The *Conditional UI* rule does not even identify a single site through statically crawled material that is not also identified by the dynamically crawled equivalent. In combination, figs. 13 and 14 make it obvious that the method of dynamically crawling the web is superior for every detection rule, even though section 6.2.2 revealed that the static crawlers collected more HTML documents per site. For reference, table 12 lists the exact amounts of matches per detection rule and crawler type.

Rule	Both	Static	Dynamic	Total
Conditional Mediation Available (Regex-based)	253	10	814	1,077
Conditional UI (DOM-based)	2	0	70	72
Navigator Credentials (Regex-based)	797	45	11,848	12,690
Password Input (DOM-based)	36,675	2,055	98,694	137,424
U2F (Regex-based)	107	14	609	730

Table 12: Matched Sites Per Rule and Crawler

6.3. Quantitative Validation of Matches

After analyzing the matches in the crawled content, this section makes an effort to verify the successful detection of authentication methods by comparing the results with a validation dataset. It is important to note that the described method only considers true positives and false negatives, as there is no suitable validation data available for false positives or true negatives.

6.3.1. Validation Datasets

To test detection effectiveness, a labeled validation dataset was required. Unfortunately, no neutral and authoritative dataset on the usage of authentication methods by specific websites is publicly available, which was one of the motivations for this thesis. To be able to perform some result validation anyway, two separate lists of websites that supposedly use some form of phishing-resistant authentication were evaluated.

Security key vendor *Yubico* publishes an online *Works with YubiKey catalog*[27] that lists compatible software products from partnering companies. For every software product, the supported "security protocols" are listed, for example, *U2F* and WebAuthn, but also TOTP and *OpenPGP*. Of course, the product list is specifically tailored to the use of hardware security keys and tends to avoid information that could reflect poorly on the vendor's products. For instance, the catalog does not provide any information on the employed authentication scheme, i.e., whether the security key is used in an MFA sign-in flow or as a single *passwordless* factor. This makes *Yubico*'s catalog unsuitable to validate the crawlers' *passkey* classifications. Moreover, numerous listed software products do not refer to a web-based online service whose website could be

[27] Available at `https://www.yubico.com/works-with-yubikey/catalog/`

crawled to detect authentication methods. For example, the catalog contains several enterprise software products operated on-premise or using tenant-specific domains. Other listings refer to browsers, operating systems, and other client software. Filtering out all unsuitable listings to make *Yubico*'s catalog usable as a validation dataset would thus require significant manual effort.

Regarding the second list, *AgileBits*, the company behind password manager *1Password*, recently launched `passkeys.directory`, which is supposed to be a community-driven index for websites that offer passkey support. As the site is fairly new and passkey adoption across the web is still in its infancy, the amount of listed websites currently is quite low. Naturally, this limits its usefulness as a validation dataset. However, `passkeys.directory` does provide a distinction between MFA WebAuthn support and the support for single-factor passkeys. Moreover, the number of unsuitable software products is significantly lower compared to the *Works with YubiKey catalog*.

In both cases, it was necessary to implement custom scraping software to extract content from the respective APIs and reorganize it to produce a practical, unified validation data structure. The implementation to collect and restructure validation datasets was done to allow for encapsulated custom provider components to be plugged into the overall process. Ultimately, the dataset provided by `passkeys.directory` was used to evaluate the detection effectiveness of the crawlers because of the reduced effort involved in verifying listing correctness and its distinction between WebAuthn-based MFA and single-factor passkey authentication.

6.3.2. Comparing Matching Rule Effectiveness

Out of the available 50 entries in the validation dataset scraped from *passkeys.directory*[28], 34 domains were visited during the crawl. Out of those, it could be manually verified that either generic WebAuthn-based or *passkey* authentication was employed on 27 websites. The remainder were authentication software products that have no sign-in process on their own websites like *Okta*, not available based on the geographic region, only accessible through a smartphone app but not over the web, or incorrectly listed.

From the 27 verified sites, 14 were correctly identified as either supporting generic WebAuthn or *passkeys* in the analyzed crawl data. This implies a detection rate of 51.85 %. Due to the size of the validation dataset, this assessment only has limited meaningfulness. Nonetheless, detection effectiveness should be evaluated to allow for improvements in future iterations. Section 6.4.1 analyses why the crawlers were not able to detect WebAuthn usage on 48.15 % of the manually verified sites in detail.

6.4. Qualitative Analysis

After statistically analyzing the crawler's collected content corpus and trying to validate matches based on publicly available labeled data automatically, this section examines the matched sites, especially the false negatives determined in the proceeding section.

6.4.1. Undetected Sites

To find possible avenues for future improvements, this section analyzes the responses fetched from the 13 missed sites from the validation dataset. For reference, these sites are `google.`↩

[28] Fetched on June 15, 2023

↳com, microsoft.com, cloudflare.com, shopify.com, nvidia.com, gitlab.com, bestbuy.com, namecheap.com, hyatt.com, virginmedia.com, mangadex.org, marshmallow-qa.com, and omg.↩ ↳lol.

Subsequently Requested JS Resources

The websites of *Google*, *Cloudflare*, and *Namecheap* all split their JS resources into chunks that are loaded only on pages where they are needed, which can save client bandwidth and improve load times. The sign-in flows of all three services were manually analyzed to find the resources that contained relevant JS code for WebAuthn-based authentication. All three sites only supported MFA at the time of the crawl, and any JS code interacting with the WebAuthn *Credentials* API was only loaded after the first authentication factor, i.e., a combination of email address and password, had been verified. Because it is unfeasible for a web crawler to create accounts on every visited website to analyze their sign-in process, subsequently requested JS resources remain a significant blind spot for detecting WebAuthn usage.

After the crawl had already concluded, Google announced that accounts could now use passkeys as a sole authenticating factor. The reflecting code changes on the sign-in page were analyzed at the time of writing. The new JS resources were found to be detectable by the *Conditional Mediation* matching rule. In an updated crawl corpus, *Google*'s use of phishing-resistant authentication would thus be correctly detected.

Subsequent Redirects and Bot Detection

Another reason why the crawler could not detect WebAuthn and passkey support on some websites is subsequent redirects on the sign-in page. Although the rationale is not always clear, this is one way of deterring web crawlers and other bots. For instance, *Shopify* and *GitLab* use a JS-based approach to trigger subsequent redirects after the page has already finished loading. In this case, the crawler saved the served HTML document and continued to the next task, as it was unaware of the delayed redirect. Aside from increasing the waiting time on every visited page, which would drastically increase overall crawl time, one approach for improvement could be trying and detect if a page intends to redirect the user after they have waited for some time.

While it remains unclear whether *Shopify* uses redirects for bot detection since their redirect delay is insignificant, *GitLab* embraces the *waiting room* approach to detect bots. Whenever a new client visits the login page at `https://gitlab.com/users/sign_in`, a dedicated waiting screen appears, saying "Checking your browser before accessing gitlab.com". It may take several seconds, during which *GitLab* presumably performs some form of browser fingerprinting, before the user is redirected to the sign-in form.

Hyatt's homepage was entirely inaccessible for both types of crawler, as it successfully detected the automated visit and served an HTML document containing only an inline script with a `429` *Too Many Requests* HTTP status code. The script could be traced back to the bot prevention vendor *Kasada*[29]. Presumably, the script is only served to suspicious clients and represents an invisible challenge that fingerprints the client and tries to entirely prevent any automated site visits.

While these sites are not in the validation dataset, similar script or `iframe`-based blocking that could be traced back to *Kasada* was found for other sites in the crawl corpus. Among others,

[29] `https://www.kasada.io`

Weebly, *GoDaddy*, *Twitch*, and *Square* all served similar challenges with a `429` HTTP status code. In total, the crawler instances have received `429` HTTP status codes from 712 unique sites, from which a small sample of responses was analyzed. Not all responses with `429` status codes are necessarily related to *Kasada*'s bot prevention. However, in every instance of *Kasada*-based blocking that could be found, a `429` status code was present.

The site `marshmallow-qa.com` showed differing behaviors between the two crawler types. While it detected the static crawler as a bot and refused to serve any content, the site did not respond until a timeout occurred when visited by the dynamic crawler. Upon inspecting the responses received from `marshmallow-qa.com` and *GitLab*, some embedded JS seems to be shared among the two sites. This shared code suggests a common origin: CDN provider *Cloudflare* appears to have blocked the requests from reaching their final destination. It is possible that the subsequent request was also blocked because an automatic visit was detected. The sites or their CDN provider *Cloudflare* could also have blocked the crawler VMs shared IP network after the initial visit.

Lastly, website provider `omg.lol` served an empty HTML document to the dynamic crawler, which may also indicate some bot prevention method being utilized. However, the static crawler was able to crawl the site successfully. Unfortunately, as the static crawler cannot execute JS, it missed an inline JS import within an HTML document that requested an additional JS file containing multiple relevant WebAuthn API calls that could have been matched in the subsequent analysis.

Insufficient Detection Rules

In the case of *Nvidia*'s website, the use of WebAuthn could have been detected with an additional matching rule. While some sites test for the existence of the *Credentials* API property on the browser's global `navigator` object, *Nvidia* and possibly others test if the `PublicKeyCredential` is present on the global `window` object. This method of detecting a browser's WebAuthn support was not anticipated before performing the crawl, but could be easily added to increase detection rate.

JavaScript-Based Location Changes

With the increased adoption of client-side rendered SPAs, some developers cease to rely on traditional HTML hyperlinks and replace them with forced browser location changes triggered using JS. This anti-pattern hurts UX and should thus be avoided. It also prevents web crawlers from accurately identifying links to other pages.

Listing 11 shows two examples of JS-based hyperlink alternatives from *Virgin Media* and *MangaDex*. The code has been shortened and formatted to improve readability. In line 2, the referenced hyperlink of the anchor tag contains what is essentially the JS-equivalent of a *No Operation* instruction. When the user clicks on the link, JS intercepts the event and executes some code that replaces the browser's currently active URL with another one. The second example in lines $8 - 12$ follows a similar approach. However, an HTML button is used instead of an anchor tag.

Without trying to reverse-engineer a site's JS code or emulating every possible event on each page, a static web crawler has little chance to detect these JS-based location changes. A dynamic crawler could possibly find the linked pages by visiting the website and actually clicking the

HTML elements it deems relevant. However, this would result in a different process design in which URLs could not be deduplicated and visited after the home page tab was closed.

Listing 11: JavaScript-Based Hyperlink Examples

```
<!-- Excerpt from virginmedia.com -->
<a _ngcontent-qnp-c108="" class="..." href="javascript:void(0)">
  <i _ngcontent-qnp-c108="" class="..."></i>
  Sign in to My Virgin Media
</a>

<!-- Excerpt from mangadex.com -->
<button data-v-623b4d2a="" data-v-9542ab21="" class="..." style="..">
  <span data-v-623b4d2a="" class="..." style="...">
    Sign In
  </span>
</button>
```

Geographical Region-Based Behavior

As today's Internet becomes increasingly splintered into region-specific fragmentations, a web crawler's point of access becomes more influential. For example, *Best Buy* places a landing page visible in fig. 15 in front of its website if the request comes from an IP address associated with a German Internet Service Provider (ISP). As the implemented crawler was narrowly scoped, it could not identify any relevant links on the landing page in such a case and moved on. This problem could be solved by architecting a less-scoped crawler that visits more or all linked pages, which would, however, require significantly more crawl and storage resources and increase the load on all crawled websites.

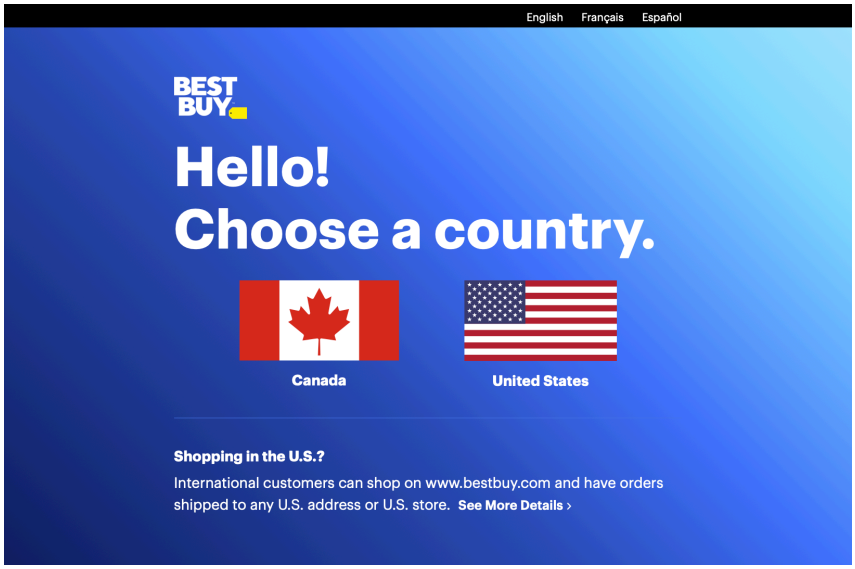


Figure 15: Best Buy Landing Page

Yahoo's Japanese website `yahoo.co.jp` that is listed on `passkeys.directory` is another more extreme example of geo-based website behavior. The site serves a block notice for visitors within the European Economic Area and the United Kingdom, informing them that *Yahoo*

Japan stopped serving their website to this geographic region in 2022 [191], effectively locking them out completely. The crawler was hosted in Germany, so it could not access the site. Because of the site-wide geoblocking, `yahoo.co.jp` was excluded from the validation dataset.

Unknown Reasons

In some cases, the reason behind unsuccessful matching remains unclear. One example is *Microsoft*'s website. When examining the fetched responses and comparing them with the excerpts from a previous manual visit, it became clear that there was an important difference between the HTML documents, which are contrasted in listing 12. The code in the listing is shortened and formatted for readability. For unknown reasons, the JS rendered different a DOM on the manual visit, which is visible in lines 12-24. While this response contains an anchor tag that could be matched using the crawler's content analysis, the fetched response in lines 1-9 does not appear to include any interactive elements. The JS of the crawled version might have an event listener that triggers a redirect, similar to listing 11, or the rendering of the inner anchor tag was delayed intentionally or due to poor performance, and the crawler had already stored the response by that point. Either way, the exact reason why the link was not rendered remains unclear. If it had been rendered, the crawler would have been able to extract and match the referenced URL. In fact, an excerpt from a manual visit is part of the unit tests for the crawler's authentication URL matching.

Listing 12: Excerpts from Visits to `microsoft.com`

```
1  <!-- Excerpt from Crawl -->
2  <div id="meControl" class="c-me"
3    data-signinsettings="{...}" data-m="{...}">
4    <div class="msame_Header">
5      <div class="msame_Header_name st_msame_placeholder">
6        Anmelden
7      </div>
8    </div>
9  </div>
10
11 <!-- Excerpt from Manual Visit -->
12 <div id="meControl" class="c-me"
13   data-signinsettings="{...}" data-m="{...}" ...>
14   <div
15     class="mectrl_root mectrl_theme_light_header" ...>
16     <a id="mectrl_main_trigger" ...
17       href="https://www.microsoft.com/rpsauth/v1/account/SignIn?ru=https%3A%2F%2Fwww.
     ↪microsoft.com%2Fde-de%2F">
18       <span class="mectrl_screen_reader_text" ...>
19         Bei Ihrem Konto anmelden
20       </span>
21       ...
22     </a>
23   </div>
24 </div>
```

6.4.2. Analyzing Matches

After investigating why some sites in the validation dataset could not be matched, this section aims to scrutinize the match statistics from section 6.2 regarding duplicate counts and rule-specific misclassification.

Inflated Numbers

Given the low number of websites listed on `passkeys.directory`, it would be surprising if every one of the 1,122 sites identified by the dynamic crawler in fig. 13a already implemented passkey support for their users. Unfortunately, the following analysis shows that this number is highly inflated by false positives where third-party-served, passkey-related JS code shared among many sites is imported but never actually used.

After identifying JS resource URLs that repeatedly occurred in the dataset and were used by multiple sites, the URLs were reduced to their hostnames, which were mapped to IP addresses. The dataset was then grouped by IP address to find all target domains and content URL hostnames that shared the same third-party JS.

The largest group of 547 sites, including big brands like *bose.com*, *marriott.com*, *nfl.com*, and *uefa.com*, all use JS from *cdns.gigya.com*[30] that matches the *Conditional Mediation* rule. *Gigya* was a customer Identity and Access Management (IAM) platform provider that was acquired by *SAP* in 2017 [165]. Today, *Gigya* is part of *SAP*'s customer IAM service used by many enterprises like the ones mentioned above. Unfortunately, WebAuthn API usage in the *Gigya* scripts does not mean that all mentioned brands support passkey authentication. In fact, none of the tested sample sites support passkeys yet. Rather, *SAP* presumably implemented passkey authentication – or is in the process of doing so – to enable their enterprise customers to support it. Due to the fact that passkeys were introduced only a short time ago, it is reasonable to assume that very few or none of these sites have enabled support yet, as the *Conditional UI* rule did not match on any of the sites using *Gigya* scripts.

The second largest group are 271 individual sites like *alphapaw.com*, *magnoliabakery.com*, and *sockprints.com* that share common JS resources, which also match the *Conditional Mediation* rule, indicating passkey support. Indeed, all of these sites are presumably hosted by *Shopify*, which recently enabled passkey support for all merchants on their platform [120]. However, in a random sample of 20 sites, none had passkey support enabled. Instead, they either used traditional passwords or login links sent via email. Thus, merchants likely have to manually enable passkey authentication on their websites, which is unfortunate given that many sites used *Shopify*'s unified authentication service and UI.

While 54 individual domains share JS resources from the `content.r9cdn.net` CDN, they were confirmed to have passkey support. However, the domains are all localized and operated by *Kayak Software Corporation*, which serves *kayak.com*, *momondo.com*, *cheapflights.com*, and *swoodoo.com* in various countries with localized Top Level Domains (TLDs).

41 individual sites used the same JS resource hosted on `www.amazon.com` that also matched the *Conditional Mediation* rule. Some of the initially visited sites had *Amazon* domains with localized TLDs. While *Amazon* has not yet publicly announced passkey support, it is plausible they would be testing whether clients supported conditional mediation to evaluate the number of potential users.

[30] or regionalized subdomains like *cdns.us1.gigya.com*

However, other sites seem to have slipped into the dataset because they set hyperlinks to *Amazon* where authentication keywords were contained within the URL. On the linked pages, *Amazon* in turn included their JS containing *Conditional Mediation* queries. Section 7.3.2 discusses these URLs in detail. In total, there are 15 sites that were wrongfully identified as having passkey support based on their link to some *Amazon* page. This incident may reveal a larger problem with the laxness of ensuring URLs discovered on a home page actually belong to the same site. Section 7.4.2 discusses possible approaches for reducing false positives through lax URL discovery.

Similarly to *Amazon*, 29 sites shared JS resources that were largely localized *PayPal* domains, with some unassociated sites mixed in that presumably linked to *PayPal* on their home pages.

26 domains using JS resources from `accounts.zoho.eu` seem to be operated by customer relation management software vendor *Zoho*. Many domains instantly redirect to a subpage under the `zoho.com` domain. These redirection domains thus needlessly inflate the number of discovered sites supporting WebAuthn. One way to combat this would be to check whether the final HTTP location after redirects was already in the crawl corpus, and if so, discarding the newly visited domain.

Nonetheless, there are sites sharing JS resources that actually all support passkeys. For example, a total of 24 websites of local banks in the United States like the *Bank of Missouri* and the *Meade County Bank* appear to employ a common digital banking platform whose JS code could be traced back to its vendor *Jack Henry*[31]. Out of a random sample of ten sites, all banks used the same sign-in UI with passkey support, customized with their branding color and imagery.

JS vs. HTML Matching

A significant fraction of matches derived from parsing included third-party JS resources may suffer from an overly lax URL discovery policy. This could apply to all authentication methods detected through JS-based rulesets. Section 7.4.2 discusses possible improvements.

However, at least in theory, detected methods derived from HTML document matches are not susceptible to the same problem, as the visible site is analyzed directly. If *Conditional UI* code is present, it is very likely that the site actually allows passkey authentication. If WebAuthn-related JS code is contained in some included script, it does not imply the code is actually executed on the site.

Because the crawler only identified a total of 72 sites where *Conditional UI* was detected, it was feasible to examine manually all of them. Unfortunately, upon closer inspection, it was clear that a significant number of domains were simple redirects to a small number of sites that were also present in the dataset. Of the 72 sites, 31 immediately redirected the crawler to a subpage of customer relation management platform vendor *Zoho*'s website, or, in rare cases, the website of one of its associated brands. Although it is unclear why *Zoho* employs domains like *zohocampaigns.com*, *zohoassist.com*, and *zohonotebook.com*, it may be related to marketing activities.

An additional 15 sites served their own content but relied on one of the other sites included in the dataset to authenticate users. In most cases, businesses use *Zoho* and allow their customers to sign in on *Zoho*'s platform. While this is debatable, it can be argued that these sites have passkey support by extension through a third party.

[31] `https://banno.com`

It could be manually verified that 20 sites have support for passkey authentication. Of those, 11 sites appear to be local banks in the United States that employ the previously mentioned digital banking platform. The 13 remaining bank websites extracted using the described IP-based method were only matched by the Conditional Mediation rule, but not the Conditional UI rule. 85 % of the discovered and verified sites matched by the Conditional UI rule were not included in the validation dataset from *Passkeys.directory*.

The remaining 6 sites were misclassified due to discovered links outside of the actual site's scope, which featured a Conditional UI. Interestingly, in one case, the mismatched site *gottadeal.com* linked to a URL under *Best Buy*'s domain, which redirected to *Best Buy*'s sign-in form. While section 6.4.1 describes how visiting *Best Buy*'s home page was not possible due to a region-specific landing page, the mismatched site was considered to have passkey support because *Best Buy*'s sign-in form matched the Conditional UI rule. Section 7.4.2 discusses harnessing these unintended matches to improve detection for the redirect target website.

The qualitative analysis has shown that a large fraction of detected authentication methods is inflated by false positives. The two identified causes seem to be a lax URL discovery policy and the inclusion of unused JS code. Nonetheless, multiple sites providing passkey support not previously present in the manually compiled *Passkeys.directory* list were newly identified.

7. Discussion

This section scrutinizes the results from section 6 by highlighting various shortcomings of the crawler's architecture and implementation, as well as the selected methods for URL discovery and authentication method detection. It also describes several obstacles and how they were overcome. In addition, unexpected findings in the crawled data and possible improvements for future work are discussed.

7.1. Limitations

The chosen methodology to explore whether authentication methods can be detected automatically through web crawling (R1) and whether it makes a difference in crawling client-side rendered content (R2) has inherent weaknesses. On the one hand, the implemented detection has several deficiencies. On the other hand, dealing with binary classification categories is challenging in a context where no ground truth exists. This section examines these limitations in detail.

7.1.1. Inherent Detection Weaknesses

A narrowly scoped web crawler that tries to analyze unknown content by capturing snapshots at specific points in time has several detection weaknesses, which are discussed below.

Detect When a Page Is Loaded

As section 4.3.2 mentions, determining if a page with unknown behavior has finished, or will ever finish loading, is impossible, as it is a classic case of the undecidable *halting problem*. Therefore, it can be assumed that a web crawler is not able to collect all resources in every case before moving on to the next page. Since it is not guaranteed that every resource is fetched, relevant content may be missing for analysis.

Resource Chunking

Many sites split their JS code into several chunks to optimize client performance. This approach leads to the absence of parts of the executed JS within an authentication flow, for instance, for WebAuthn-based MFA. Instead, the necessary JS to interact with the browser's *Credentials* interface is subsequently requested once the user has successfully authenticated with their first factor. A web crawler cannot advance that far into an authentication flow without having a prepared account for each visited site and circumventing possible *CAPTCHAs*, which is not desirable behavior anyways. Thus, some authentication methods are not detectable by a web crawler if the associated HTML and JS code is not referenced on pages that can be publicly visited. However, this limitation is only relevant to MFA flows where WebAuthn is not used for initial authentication.

Websites Not Using Anchor Tags

Listing 11 in section 6.4.1 shows two examples of many possible ways to re-implement traditional hyperlinks using JS. Instead of using standard HTML anchor tags with `href` attributes containing the linked URL, some developers find creative ways to send users to linked pages.

They may intercept click events on a button element, or a plain `div` tag. Without trying to reverse-engineer the JS code, it can be very challenging to discover these kinds of page links automatically.

Miscellaneous Link Labels

Even if a traditional hyperlink is used, the studied crawler is narrowly scoped, so it has to decide whether to follow any discovered links. This decision is based on two pieces of information: the linked URL itself and the link's visual label. Since there are lots of ways to paraphrase the link text for a sign-in page, it is likely that the list of known phrases is not all-encompassing and some pages will thus be missed. For example, the only link to access one's account on *Apple*'s home page reads "Manage Your Apple ID". On other sites, the sign-in link may be only labeled by a visual, like an image or an icon representing a user without any descriptive text. It is thus reasonable to assume that the crawler misses some sign-in links that may be obvious to a human looking at the rendered page.

Language Bias

Another limitation is the implementation's inherent language bias. The crawler can detect login keywords in 58 languages, provided the website provides an HTML language tag. However, the localized keywords could only be composed manually for a small subset of languages. The rest was automatically generated using a combination of the publicly available services *Google Translate* and *DeepL*. Especially for languages that appear less frequently on the Internet or are spoken by fewer people, it is possible that those keywords do not represent the correct terms that are genuinely used in the respective language.

Additionally, localized URLs are not considered when deciding on discovered URL relevance. For instance, the English URL `https://example.com/login` would be matched, but the localized German URL `https://example.com/anmelden` would not.

Geofenced Content

Sites serving regionally-specific content or outright blocking some regions from accessing their content constitutes another limitation that was observed during the crawl, as section 6.4.1 discusses. Because the crawler VMs were hosted within Germany and had Internet access through a German ISP, they did not have the same view on the web that a crawler in the United States, Japan, or China would have. In an increasingly fragmented Internet, one's geographic location restricts the reachable and thus examinable content.

A/B Testing and Fluctuating Content

Since the crawler only captures a single snapshot at a specific point in time per site, it may be subject to *A/B testing*, temporary content changes, or outages. For example, a website may not serve the new version of their authentication page to the crawler because it is not assigned to the respective cohort that the new version is tested on. Some vendors may also link the testing of new features to a geographic region, effectively creating a stable cohort for behavioral analysis.

Lax Link Discovery Policy

Some operators have unified Single Sign-On (SSO) platforms that may be available at a website's subdomain, while others use an entirely different domain name. Some sites use third-party platforms for authenticating users and making personalized or protected content available to them. This effectively splits sites into parts that are publicly available and others that are not. Different providers on different domains may host these two parts. To cover all of these use cases, the crawler follows links on visited home pages containing keywords in the URL or the link text, regardless of the link's final HTTP location. This lax policy has the disadvantage of making false positives more likely. Unfortunately, this behavior is one of the main reasons why many false positives are identified in the crawler's matches.

A related issue is the question of how to deal with third-party sign-in options like the widespread *social login* offerings where users authenticate using their *Facebook* or *Google* accounts. Whether passkey support provided by third parties should be included in a dataset is debatable. For this thesis, an exclusion of third-party authentication was not considered.

7.1.2. False Negatives vs. False Positives

While it cannot be guaranteed that every site using a WebAuthn-based authentication method is detected, lowering the number of false negatives is the easier of the two misclassification categories. To optimize the crawling and content analysis components, multiple parts of the implementation are unit tested with constructed environments, as well as real-life excerpts from sites that are known to have support for the tested method. In a subsequent step, the matches generated from a real web crawl corpus can be validated using labeled data, as section 6.3 describes.

Lowering the number of false positives is more complex because it involves manual effort in verifying results. Unit tests covering the investigated cases could be added to the implementation. However, reducing the number of misclassifications on other sites based on matching code that exists in a website's JS code, but is never executed, is significantly more challenging than adding new matching rules to cover previously unconsidered indicators of support. Differentiating between legitimate SSO platforms hosted on other domain names and unrelated web services is also difficult. Furthermore, no dataset exists listing websites not supporting particular authentication methods, which could be used to validate crawl data matches automatically.

7.1.3. True Positives vs. True Negatives

Besides validating results automatically using labeled data, verifying a true positive classification is possible by manually investigating the website in question. If the crawler was able to detect some pattern in the publicly available HTML or JS, manual examination should yield the same result. The same is not true for true negatives, as it is impossible to prove that a site does not use a particular authentication method. As section 7.1.1 mentions, sites may split their JS resources into chunks and serve the relevant HTML and JS portions only in some arbitrary situation, for instance, after a user successfully authenticated with their first factor in an MFA flow. Due to the potential for such unpredictable behavior, it is not possible to determine that a site is not using a specific method with absolute certainty.

7.2. Overcoming Hurdles

This section examines numerous obstacles encountered during implementation, deployment, crawling, and analysis and describes how they were overcome.

7.2.1. Infrastructure

When deploying the designed system and performing the web crawl, several infrastructure-related issues occurred.

IPv6-related Issues

Because of the deployment environment's quirks, numerous problems occurred that would presumably not have arisen in another environment. As outlined in section 5.3.1, the OpenStack tenant had access to a single dual-stack network with a private IPv4 address and a public IPv6 address. The IPv4 network was present, but was not able to route any egress packets. At the same time, the tenant had restrictions in place that prohibited creating software-defined internal networks and routers, which is one of OpenStack's core functionalities.

The first challenge related to the described broken dual-stack network stack was to get elementary OS components to work properly. To install any software, the package managers needed to be able to update their indexes and download packages from the Internet. As some signing keys required for downloading packages from custom `apt` repositories were only accessible via IPv4 on `keyserver.ubuntu.com`, installing all packages required setting up a NAT64 environment with DNS64 address translation, as detailed in section 5.3.2.

It had been initially planned to deploy containerized software components using Docker images and Docker's container engine. After performing several tests, experiencing odd behavior, and studying the available documentation closely, it became apparent that Docker's experimental IPv6 support [57] caused more issues and additional effort than the benefits of containerization were worth. After all, the deployment on dedicated VMs was automated with Ansible anyways, so running the software directly at application level on the VMs made sense.

While RabbitMQ has decent documentation on the networking options required to listen to IPv6 interfaces [186], working out IPv6 connectivity for Java-based Cassandra and its monitoring companion JMX Exporter was more challenging. Because no proper documentation could be found, the first approach was to filter all `A` record responses out of the NAT64 gateway DNS server's responses using *BIND 9*'s `filter-a` module. As *BIND 9* also created and served NAT64-based `AAAA` records for any IPv4-only hostnames, this could have solved the issue of software preferring IPv4 addresses if both options were available. Unfortunately, the `filter-a` module seemed to have been applied before DNS64, which resulted in IPv4-only hostnames not getting filtered. The second attempt was to modify the Ansible deployment, removing all IPv4 routes from the VMs and disabling IPv4 connectivity entirely. While this helped with some of the overall experienced connectivity issues, Cassandra and JMX Exporter were still not communicating with other hosts. After more research, it turned out that the Java Virtual Machine (JVM) for both components required the additional startup option `-Djava.net.preferIPv4Stack=false`. While this change allowed the Cassandra instances to finally communicate, the JMX Exporter still was not able to bind to any interfaces. Ultimately, it turned out that JMX Exporter needed the additional JVM startup option `-Djava.net.preferIPv6Addresses=true`.

Logs Filling Disks

After running one of the preliminary stress tests over an extended period of time, it became apparent that the default OS log rotation policies could become a problem with the intended use case. Figure 16 shows how the crawler VMs filled their disk throughout the stress test, until the instances became fully unresponsive. The crawler's `systemd` service had filled the `syslog` file until the default-sized 12 GiB VM root disk was full. Because the default `logrotate` cron job only runs daily and the default `rsyslog` configuration rotates `syslog` weekly, retaining four versions, the crawlers would have potentially filled their root disks during a long-running crawl operation, too. Hence, the `rsyslog` and `logrotate` cron job configurations were modified to rotate logs hourly based on file size instead of creation date. Retaining old versioned log files was disabled and the file size limit was set to 100 MiB. Fortunately, this limitation was already discovered through testing before beginning the extensive web crawl.

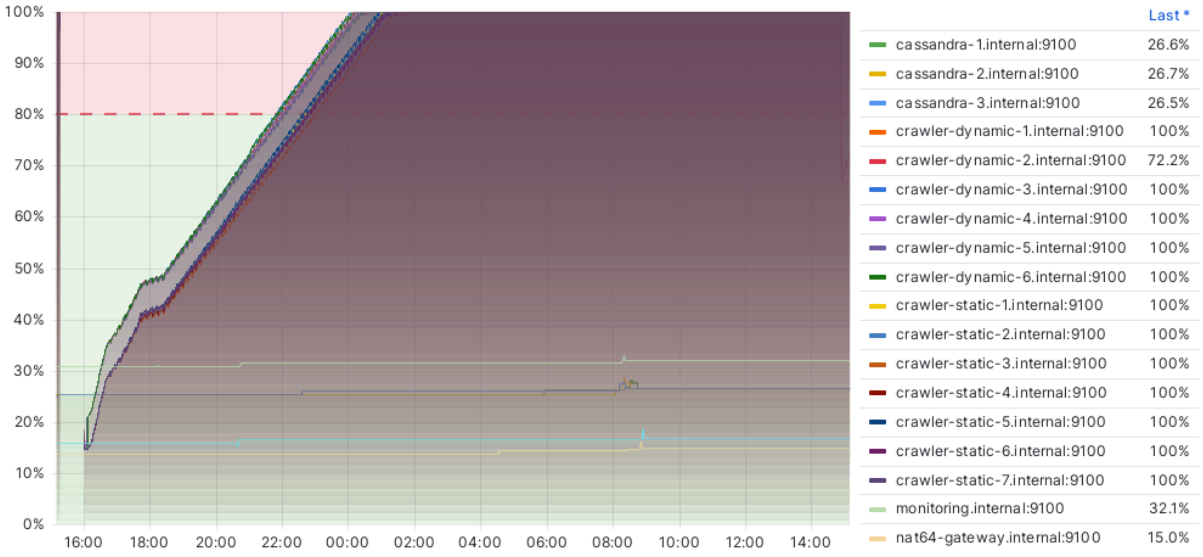


Figure 16: VM Disk Usage During and After Stress Test

Debugging a Distributed System

When debugging the first complex issues that included multiple instances and were partly timing-related, it became evident that debugging a distributed system is quite challenging, even though monitoring tools instrumented all relevant components. What could have helped is a unified logging solution that would have allowed tracing errors over multiple instances and software stacks. However, the cumulative amount of logging data would have required additional storage resources, which were limited by a fixed quota in the deployment environment.

7.2.2. Browsers and their Complexity

As section 6.1.2 describes, remote-controlling software as complex as a full web browser is not trivial. This section examines some of the issues that arose in relations to integrating Chrome into the crawling process.

Debian's Chromium Patch to Disable SwiftShader

While working to optimize Chrome's stealthiness to avoid being caught in some CDN's or website operator's over-blocking of automated visits, which section 4.5 details, a renderer-related issue occurred on the deployed VMs' Chromium. It was not reproducible on a personal client running both Chromium and Google Chrome on macOS. Basically, the browser's *WebGL* 3D rendering API was not working, which caused the browser's bot detection results (fig. 3) to be suspicious. After extensive research, it turned out that the Chromium package in Debian's standard repository ships with a manual patch disabling *SwiftShader* support [73]. Because *SwiftShader* was not available, there was no *WebGL* driver available. Thus, *WebGL* support was missing, which was flagged as suspicious client behavior. Ultimately, the issue could be resolved by installing Google Chrome from Google's `apt` repository, replacing the standard Chromium package. Google's browser did not include any patches that would disable *SwiftShader*, which resulted in proper *WebGL* support.

Dying Workers

During the crawl, the number of dynamic crawler workers started to continuously decrease at some point after starting the crawler service on the instance. While other causes could remain undiscovered, one reason became clear after investigating the crawler's logs. If a single page load was not completed within five seconds, or the entirety of a targeted site was not crawled within 20 minutes, a timeout was triggered to prevent wasting time on exceptionally slow websites and to protect against *crawler traps*, which are explained in section 2.2.1.

If a long-running operation should be aborted in Go, its context can be canceled. The operation then aborts and returns a *context canceled* error. Normally, cancellation propagates to any child contexts, but not to the parent context. It remains unclear why this happened, but when the request operation timed out, and its context was canceled, `chromedp` closed the entire browser tab associated with the worker *goroutine*. However, a worker's main *goroutine* intentionally used a parent context to manage the state of its dedicated browser tab. When `chromedp` closed the tab, the worker was placed into an undefined state, which resulted in a fatal error on the next browser-related operation, where the worker ceased to process crawling tasks. As a workaround that was necessary for a disk space leakage issue discussed in the following, every dynamic crawler was restarted every 120 min, which also revived the dead workers.

Chromedp Disk Space Leakage

After discovering that log files could potentially fill the crawlers' disks, as discussed in section 7.2.1, the VMs' disk usages were closely monitored when starting the crawling process. While the static crawler instances had no problem with increasing disk usage, something seemed to hog disk space on the dynamic crawler instances. After examining the file system of several VMs, it was found that a considerable amount of Chrome and `chromedp`-related files in the machine's `/tmp/` directory had accumulated. The files followed the two patterns `/tmp/.com.google.Chrome.JxpSF6` and `/tmp/chromedp-runner1324742760`, where the suffix represents a unique identifier that differed for every file. The former represents the user data directory for every started Chrome process mainly used to store cached data, while the latter appears to be a `chromedp`-related file cache.

Unfortunately, this disk space leakage seems to be a bug in the `chromedp` library and was mitigated using the previously mentioned workaround of stopping every dynamic crawler instance

every 120 min, performing a clean-up operation that deletes all residual temporary files, and restarting the workers. The duration was chosen based on the observed leakage rate.

Chromedp Event Listener Race Conditions

During the crawl, several errors occurred where the crawler was not able to receive the bytes of resource responses received by Chrome. On any dynamic crawler visit, the page's DOM was extracted from the browser directly, while any additionally loaded resources like JS files were intercepted using the network tracing *DevTools* API and received through events.

Unfortunately, the *EventResponseReceived* event, which contains the requested URL and HTTP response status code, does not guarantee that the response body is readable yet. The *EventLoadingFinished* event guarantees readable response bytes, but it does not contain any reference to the requested URL and HTTP response status code. Thus, the crawler had to listen to both events to combine the two sets of information for every fetched response. After navigating to the targeted page URL, the crawler waited until all concurrently running event handler *goroutines* had finished processing events. However, because the delivery of events occurs asynchronously, the *EventLoadingFinished* event sometimes was only triggered after the page visit was already concluded, and the worker had moved on to its next crawl operation. When the event handler processed the old event, it was not able to access the response bytes, which produced the observed error.

The solution for this problem was to implement an atomic counter that was increased on every *EventResponseReceived* event. Whenever a *EventLoadingFinished* event was received, the counter was decremented. The main worker *goroutine* then blocked until the counter was reduced back to zero, indicating that every *EventLoadingFinished* was received. Additionally, a timeout was added to prevent potential deadlocks over the counter in case an error occurred in Chrome's event processing. After the counter stopped blocking, the worker was blocked as before, waiting for every event handler to finish.

Auto-Accepted Downloads

Like with Chrome's temporary files, the browser's default behavior of auto-accepting file downloads when the `Content-Disposition: Attachment` HTTP header is present also caused some of the crawler instances to fill up their disk. It had not been considered that the default behavior would be to accept and store downloads in the user's home directory. Fortunately, a `chromedp` configuration parameter for starting Chrome can be used to set the default download behavior to `Deny`, which resolved the issue.

7.2.3. Message Broker Complexity

Chrome is not the only piece of complex software within the crawler's architecture. For instance, when working on the crawler software and implementing concurrent task processing, only one worker *goroutine* did any work. This behavior had two individual causes. First, the initial draft used one *AMQP* channel to communicate with RabbitMQ per crawler instance. However, a single channel is not supposed to be used concurrently, which caused all other *goroutines* to block when the first *goroutine* used it to consume a queue.

After refactoring the code to open one unique channel per *goroutine*, all workers still idled except for one. It turned out that RabbitMQ limits prefetching to 250 messages by default to optimize

performance, which caused the first worker to hog all available testing tasks. This was not a huge issue – but it highlighted a potential problem with preserving the crawling order. As the number of pages that could be crawled was not known beforehand, the order of domains on the targeted popularity list should be preserved to ensure increasing coverage without any gaps between domain ranks. If the default prefetch count were used, there was a potential for significant gaps between ranks when stopping the crawl without completing the entire list of targeted domains, which is what ultimately happened. To roughly estimate the maximum gap size, let the number of static crawler instances be 5 with 30 concurrent workers each and let the number of dynamic crawler instances be 10 with 25 workers each. Equation (5) shows that a prefetch count of 250 could result in a gap in the list of crawled sites of up to 100,000 domains. Thus, prefetching was disabled to preserve the rank order in the crawled data.

$$(5 \times 30 + 10 \times 25) \times 250 = 100{,}000 \tag{5}$$

7.2.4. The Wild West of the Web

Because of the various ways websites are built, often forgoing standards and best practices, parts of the web resemble a place without rules, like the Wild West.

Data Scheme URLs

Typically, hyperlinks on the web point to a URL that users can visit by clicking on the associated anchor tag's label. Section 7.1.1 discusses that some developers replace traditional hyperlinks with event-based JS magic to redirect users. However, the opposite scenario also exists, where an anchor tag has a `href` attribute, but it does not contain a URL. Instead, a different scheme is used to repurpose the anchor tag. There are benign examples, like using the `tel:` or `mailto:` schemes that are used to allow users to call a phone number or compose an email to a specific address. But it is also possible to place arbitrary data in URI attributes using the `data` scheme. This has legitimate use cases, like inlining a small picture into an image tag's `src` attribute to avoid an additional HTTP request. Theoretically, a data scheme URL can contain any type of data. To identify the content type, a MIME type can prefix opaque data. Apparently, some website operators use this fact to place JS code into an anchor tag's `href` attribute that is executed when the user clicks on the anchor tag. This behavior was discovered during the crawl because the crawlers tried to visit these URLs, and the issue was mitigated by filtering the discovered URLs by their scheme. Nonetheless, a total of 68,188 unique data scheme URLs from 9,192 sites were found in the resulting content corpus. While some sites appear to use this technique to track clicks, the motivation is unclear for others. Because of the high number of sites, it is conceivable that a shared JS library uses this approach to encapsulate custom click behavior. Even though no evidence of abuse was found in the examined samples, using JS in an anchor tag's `href` attribute can also be used by malicious actors to perform cross-site scripting attacks.

Detecting Content Types

Not all links on a website point to another HTML document on the web. Sometimes, files like PDFs or images are linked instead. Also, when websites reference URLs as script sources, the returned response does not always have the expected content type. That is also true for the `Content-Type` header in a HTTP response. As a result, data of an unexpected type is

served. Trying to detect the content type of a raw byte response is difficult. For example, guessing the MIME type of a JS response containing non-ASCII characters can result in a generic `application/octet-stream` classification. When testing different implementations to guess the MIME type of fetched responses, for instance, with the `http.DetectContentType()` method in Go's standard library, it was found that the misclassification rate is so high that guessing did not provide any usefulness.

String Encoding

At the very beginning of the crawl, an error occurred where the response from Google's home page could not be saved to Cassandra with an error message saying "String didn't validate." Fortunately, this error occurred early and was discovered, as it prompted an exploration of string encoding in the first place. The reason why the content could not be saved is that Google's web server encoded the response using `ISO-8859-1`, but it was tried to save the content as `UTF-8`. An easy solution is to change the column type from string to blob, but if the content was to be analyzed and matched, it was important that no encoding errors prevented a matching rule from applying. As it turns out, some sites do not even specify which encoding they used for their response. Hence, a dedicated string transcoding component was implemented that looks up the charset from the HTTP response headers, or guesses the charset if the web server does not specify one. Before the content was stored and processed, it was transcoded from its original encoding to `UTF-8`. In case an exotic encoding that could not be transcoded appeared, the column type was changed to blob to prevent possible storing errors, nonetheless.

Response Sizes

Something that was also discovered because of a database error was the enormous amount of data some websites serve in the form of singular HTML or JS responses. The related database limitations are discussed in section 7.2.5.

The latest *Web Almanac* report places the median page weight for HTML content at 31 kB, and at 509 kB for JS [93]. In contrast, one single JS response returned from `https://portal.forter.com/dist/app-07439ecfb5a1e2a72f0847e7ee839f859465a422.js` had a size of 28.7 MB. That is 564-times more data in a single file. Upon inspection, it became clear that the operator had shipped the development build of a mid-sized Angular SPA codebase that did not use purging unused code, minifying, or chunking. The file also contained the source code of numerous third-party JS dependencies. Interestingly, the URL shows that versioning and cache busting are used, which is unusual when not using proper production builds.

Another example is the HTML response returned from `https://clay.run`, which had a size of 20.4 MB. That is over 658-times more than the median HTML response. At the time of writing, the website serves a significantly smaller response with 763 kB in size, which indicates the operator recognized their mistake and mitigated the issue.

7.2.5. Database Limitations

As mentioned in sections 3.2.2, 6.1.2 and 7.2.4, during and after the crawl, several issues occurred that are related to the way the Cassandra wide-column store was used in the crawler architecture. This section examines these problems and their causes.

Maximum Key Sizes

After a few thousand sites had been crawled, it became evident why Bahrami et al. chose to use hash values of visited URLs as row keys [18]. This was not considered when initially designing the crawler, but many URLs linked on the web have a long list of parameters, tracking information, and opaque data attached to them, making them quite long. A text column in Cassandra has a maximum size of 2 GiB, which would allow for an excessively long string. However, Cassandra's maximum key size is 64 KiB to optimize performance, which is reasonable. Hence, using long URLs as partition keys can quickly become problematic. As a result, Bahrami et al.'s approach was adopted to mitigate the issue after realizing the mistake. Every row then used the URL's hash as a partition key and contained the unhashed URL in a simple text column since the URL is essential for subsequent content analysis.

Maximum Request and Commit Log Sizes

The size of responses saved to Cassandra were perhaps the most drastic problem in the crawler's architecture. As described in section 7.2.4, some sites serve excessively large amounts of data. Storing these large responses revealed some of Cassandra's limits. The documentation states that it is possible to store blobs of up to 2 GiB, although it recommends to keep them under 1 MiB [53]. Unfortunately, this advice was not taken seriously enough while planning the architecture. More thorough research could have discovered that, on another page, their website points out that Cassandra is not optimized for large blobs of data and that the maximum query request and commit log sizes would become a bottleneck [177].

The first error occurred during the crawl, when a larger number of small responses from one site were sent to Cassandra to be persisted using the CQL Go client's *saveMany* method. As it turned out, *saveMany* sends a single request containing multiple rows of data. The initial solution was to change the implementation to save every response individually to avoid hitting the yet unbeknown request size limit. In an attempt to optimize the duration of storage operations, several inserts were performed in parallel, as the documentation suggested. However, the Cassandra instances became unresponsive repeatedly, so the change was reverted. In hindsight, the blob sizes presumably overstrained the instances.

After discovering that some responses exceeded the maximum CQL query limit of 16 MiB and how this limit relates to the commit log size limit, which must be at least double the maximum request size, Cassandra's configured limits were adjusted, including doubling the maximum frame size to 32 MiB. The crawler implementation was also modified to discard any responses larger than 28 MiB, leaving a buffer for the additional metadata saved with the response. The tradeoff was that some sites may not be matched because their content could not be processed. Fortunately, after implementing the response size limit, no error related to the response size was logged, indicating that this restriction did not affect the subsequent content analysis.

Backup Process: Readout Speed

The real performance penalty of storing large blobs in Cassandra became apparent after the dispute with *bwCloud* personnel forced interrupting the crawl. Since, at the time, it was not clear whether the environment could still be used for data analysis, a backup process was implemented and data was migrated off-site. Initially, when trying to extract data from Cassandra using its bulk exporter, the instances crashed repeatedly. Since this approach did not seem feasible, custom implementation was added to the crawler's source code, subsequently querying paged

content for every visited domain and storing it locally on a solid state drive. Because the content corpus was already sized at multiple TiB without having completed the entire crawl, responses were compressed using Zstandard (Zstd), which is a fast compression algorithm that provides high compression ratio and was open-sourced by Facebook [50]. By using Zstd, all collected responses of the visited 624,780 sites along with metadata take up little over 1 TiB. The response metadata was saved in a CSV file per site. The logged errors were saved in an additional CSV file per site. To avoid overstraining the filesystem by having too many directories on the same level, the data folders for every site were combined using a hostname hash prefix, following a similar approach to HIBP's *k-Anonymity* buckets [90].

When querying a paginated list of the responses for a specific hostname, it became clear that Cassandra performs significantly worse when reading large blobs compared to writing. Considering that Cassandra is not optimized to handle large blobs, Figure 11d in section 6.1.2 shows that write operations performed exceptionally well with a duration of largely under 250 ms in the 99^{th} percentile. In comparison, reading responses for a hostname took 52.5 s in the 99^{th} percentile, with a mean value of 11.6 ± 11.8 s. In the 10^{th} percentile, queries took 1.6 s, the minimum duration was 28 ms. Although the queries were performed over the Internet instead of within a virtualized network, which significantly increased latency, sample queries performed on hosts within the deployment environment showed similarly poor read performances.

Thundering Herd: Backup Process

To speed up the data migration, the exporting implementation used multiple *goroutines* to concurrently query for data. The optimal number of 25 concurrent *goroutines* was determined experimentally. Numerous errors occurred during the backup process, which ran for several consecutive days. They were either related to connection issues or large blobs overstraining the Cassandra instances. The exporter performed a local presence test before sending a query to avoid duplicate queries. Initially, any CQL error triggered a simple restart of the exporter, which caused all 25 concurrent *goroutines* to request the responses for a hostname. That caused a *thundering herd problem* where the Cassandra instances were overwhelmed by the amount of concurrent requests. That caused errors, which caused more concurrent requests. To mitigate this problem, the exporter implementation was modified to slowly ramp up the number of concurrent requests, which successfully resolved the issue.

7.3. Unexpected Findings

While examining the content collected during the crawl, several striking patterns and behaviors were found in the data that are presented in this section.

7.3.1. Extensive Link Collections

When specifying the crawler, it was not anticipated that some sites would contain an extensive collection of links on their home page that matched a discovering rule for authentication URLs. The three largest link collections are explored in the following. These examples demonstrate why it is challenging to discover sign-in pages on websites without considerable overmatching.

The third-largest link collection with a total page count of 275 appeared on `elektroniczne-zapisy.pl`, which appears to be a platform where users can sign up for sporting events. Because the events' registration forms are accessed using URLs like `/event/8559/signup.html`, these

URLs were all visited by the crawler. However, not all of the 274 discovered links on the home page were falsely discovered. The URLs `/login.html` and `/registration.html` indeed corresponded to the website's user login and registration forms. Content analysis discovered a password form on both pages, which was confirmed by manual examination. The needlessly visited URLs did not skew the results, though, as no authentication method was detected on any other page.

The second-largest collection of links was found on `justdeleteme.xyz`, with a total page count of 318. *JustDeleteMe* hosts a community-curated list of links that visitors can follow to delete their account from web services, which employ dark patterns to hide account deletion options from their users. Obviously, many links on the page have relevant keywords in their URL that are matched by the crawler's URL discovery component. This is a clear example of links on a home page that do refer to some account management pages, but are not associated with their origin.

The largest total page count of 362 unique URLs was found on `dnscentral.com`, the website of a domain registrar and DNS provider. Similar to the first example, the better part of the discovered URLs on the site refer to domain registration, not user registration. The discovered URLs mostly have a pattern similar to `/registration/tld/com`. The visited URLs do not skew any results because no authentication methods could be detected on those URLs. The only match for password-based authentication appeared on `/registration/renew`, which was manually confirmed to render a sign-in form before a visitor is able to register a new domain. The site does actually have a dedicated "Account Management" link on their home page, but the response rendered a screen providing the option to sign into a domain registration account or a managed DNS account, indicating that the site may be splitting its user base by a user's utilized offerings. Because the crawler is narrowly scoped and only crawls to a depth of 1, the linked sign-in forms were not reached.

7.3.2. Amazon Links

Other interesting finds were accumulating authentication method detections that occurred on a URL under the `www.amazon.com` domain for sites that simply linked to Amazon on their home page. It can be assumed that such misclassifications also occurred for other sites in the analyzed data. As section 7.1.1 explains, this is the result of a too lax URL discovery policy. After examining several samples, it became apparent that these sites either linked to individual Amazon seller profiles or items available for purchase.

For example, the operator of `anikasdiylife.com` linked to their Amazon seller profile. These Amazon profile URLs seem to have a common pattern: `https://www.amazon.com/ideas/amzn1.account.XXX/XXX`. These URLs got matched because of the "account" keyword.

Another example is the blog `daddytypes.com`, which linked to a stroller available for purchase in one of the blog posts. Interestingly, this URL and others like it on different sites in the corpus were only matched because of a tracking parameter in the URL: `http://www.amazon.com/.../Stroller-Base-Black.../?...&ref_=nav_signin`. The `ref_` parameter presumably indicates that the user clicked on the sign-in button in the navigation bar before viewing the product. Because the blog author probably copied the URL from their browser's navigation bar, the tracking parameter was included in the link on the blog post.

7.3.3. Conditional Rendering Makes HTML Detection Rules Difficult

In terms of false positives, Section 6.4 shows that the *Conditional UI* detection rule is more reliable than JS-based rules. However, it was observed that a small number of websites only render the relevant autocomplete parameters on HTML input in specifically targeted browsers. When manually visiting sites that verifiably supported passkey authentication in Firefox, a handful did not have the `webauthn` keyword in their input autocomplete parameters, even though they included it in Chrome and Safari. This may be a reason why the static crawler was only able to identify two sites that had *Conditional UI*.

The conditional rendering of autocomplete parameters is unfortunate for several reasons. It not only makes the automatic detection of passkey support more challenging – allow-listing clients also hurts a website's usability. Users then rely on the website providers to add their browser of choice to an arbitrary allow list, even though the browser may have added passkey support without the website operator being aware. In addition, the reasoning is not comprehensible because unknown autocomplete keywords are typically just ignored by browsers. Thus, always adding the relevant autocomplete parameter does not have any disadvantages. If the browser did not support credential discovery, the user would not notice any difference in the sign-in form's behavior. It is possible that sites use the `isUserVerifyingPlatformAuthenticatorAvailable` method on the *PublicKeyCredential* interface to determine if platform-specific authenticators like *Touch ID*, *Face ID*, or *Windows Hello* are present on the client. However, this would still not make any sense for the above reasons.

In a web crawling context, this behavior narrows the choice of a web browser to use for visiting sites if maximum coverage is a goal. Basically, there is no reasonable choice besides using a Chrome-based browser.

7.3.4. Websites May Detect Failing Image Rendering

One of the measures to optimize performance for the dynamic crawler was to intercept all network traffic and block stylesheets, images, and videos from loading. While examining the crawled content, the website `wowhead.com` was found to detect whether clients successfully load their images and JS resources. Because the crawler blocked the image requests, the website redirected the client to an error page[32] at `https://www.wowhead.com/error-static-cdn`, explaining that loading their static CDN content had failed. It was thus not possible to fetch the site's actual HTML.

While no other instance of such behavior was identified, this case nonetheless indicates that optimizing performance by limiting a website's ability to load content may have unintended side effects that should be considered.

7.4. Future Work

This thesis proposed utilizing a distributed web crawler to automatically identify websites supporting phishing-resistant authentication methods. This section offers several suggestions on what could be improved in future research approaches.

[32]The page was archived at `https://web.archive.org/web/20230626145123/https://www.wowhead.com/error-static-cdn`

7.4.1. Architectural and Infrastructural Improvements

Before collecting any content or analyzing it, there are some vital improvements that could be made to the crawler's architectural and infrastructural design.

Unified Logging

Whenever errors occurred during initial tests or the web crawl, it was challenging to debug how multiple components in a distributed system interacted and to find causal relationships between individual behaviors. Even though the system was well-instrumented, examining logs that were bounded to the respective host VM was sometimes challenging. A unified logging solution would have improved this situation dramatically. Therefore, it is advisable for future work to plan central logging collection from the outset to avoid sifting through logs on all instances to find that one process experiencing errors.

Detect Database Limits Early On

Another shortcoming of the presented architecture was the database performance for large content blobs. Testing the intended use case in its entirety would have flagged potential issues early on. Unfortunately, extensive tests were only performed for the first half of the use case: storing exemplary crawled content. If reading responses from the database for analysis would have been tested in the initial implementation phase, modifying the architectural storage components would still have been feasible.

7.4.2. Improve Authentication URL Detection

Several approaches could improve the detection of authentication-related URLs and reduce the rate of false positives.

Detect Content Management System

A promising approach to discovering the URL of sign-in forms could be to detect the utilized Content Management System (CMS) and try to navigate to the CMS' default login path. As there is a small number of profoundly popular CMSs, compiling a suitable list of URLs is manageable. However, there is the question of whether the CMS login is actually the avenue used by users to authenticate. In the examined crawl dataset, multiple sites were found to use one of the popular CMSs for their website, but used a different software product to authenticate customers and let them access some kind of internal content that is not served using the site's CMS. Thus, it is possible that this approach could lead to false positives.

Improve Multi-Language Support

As section 7.1.1 describes, the crawler's implementation is able to detect localized login link labels through automatically translated keywords. This detection could be improved by using verified translations since automated results are likely to be of worse quality and use the wrong wording. For instance, the Polish sporting events platform mentioned in section 7.3.1 rendered a login link with the label "Zaloguj się", while the automatically translated Polish keywords are "zalogować się" and "zarejestruj się". This particular link was still discovered due to an English keyword in the URL. However, some sites may use localized keywords in their URLs, which the

current crawler version cannot detect. Hence, adding translations to matched keywords in URLs in addition to labels would further reduce the inherent language bias.

Ignore Unassociated Linked Content

One of the main causes for an inflated false positive detection rate was the authentication method detection on linked third-party content not associated with the examined site. When designing the URL discovery policy, first-party SSO systems on other domains, third-party "social logins", and related third-party software like an outsourced customer login were not supposed to be excluded. Thus, a policy was implemented that matched any hyperlink on the home page of a website that featured authentication-related keywords in its label or URL. However, as section 6.4 shows, this lead to numerous misclassifications. To reduce the number of false positives, a stricter policy could be applied that only considers URLs with the same base origin[33]. This would exclude the previously mentioned cases, though.

Deduplicate With Final HTTP Location

For some sites, the same content was stored multiple times because deduplication happened at URL level after collecting them from the home page. However, any sites linked to unique URLs, which all redirected to the same final HTTP location. To avoid storing content multiple times, a second deduplication step according to the final URL after any redirects could be implemented before storing the responses. Sometimes, URLs with the same origin as the visited home page also redirected to a different domain, for example, when the site used link tracking. These links would also have to be discarded with the stricter URL discovery policy. Although, the final destination's domain could be classified using the discovered URL instead.

7.4.3. JavaScript Deobfuscation

To improve the detection of obfuscated JS code, it may be possible to study common obfuscation techniques and try to reverse them on collected JS resources. There are some projects like *Restringer* [21] that try to reconstruct obfuscated JS strings by reversing common obfuscation methods like changing the encoding or splitting a string into multiple subsets and joining them back together. Listing 9 in section 4.3.2 shows a simple obfuscated call to the browser's *Credentials* interface that could possibly be deobfuscated using similar methods as *Restringer*.

7.4.4. Improve JavaScript Source Detection for Static Crawler

There are conceivable ways in which the static crawler's detection rate may be improved. Figure 12 in section 6.2.2 shows that the static crawler was not able to collect the same number of JS resources per site when compared to the dynamic crawler. Enabling it to discover more URLs referencing JS code may help to improve its performance. One way could be to analyze the JS source files referenced in the HTML and look for indicators for resource chunking. As most served JS is compiled using only a handful of module bunlders like *Webpack* [109], it may be possible to achieve relatively large coverage with little effort. A simpler method could be to match strings in the JS code that resemble a URL to another JS file and try to crawl it.

[33] generally the second-level domain

7.4.5. Deduplicate Regionalized And Redirecting Domains

In order to deflate the number of duplicate matches, domains that immediately redirect to another site that was already visited could be to discarded. During the crawl, many domains like `zohocreator.com` and `zohocrm.com` were found in the list of detected passkey support. However, these domains redirected to *Zoho*'s main website, which was then analyzed and counted multiple times. For future work, it would make sense to consider these cases and discard redirection domains. This could also be achieved using the aforementioned second deduplication step for the final HTTP location URL. What is more difficult is to deduplicate regionalized domains like `zoho.com.cn`. However, one could argue that regionalized domains that do not directly redirect to the main website should actually be considered a separate site that can be listed separately.

7.4.6. Detect Common Passkey Libraries

In recent months, several libraries abstracting passkey authentication started to gain popularity, for instance *Hanko* [22]. Another way to improve the detection of authentication methods could be trying to detect usage of one of these libraries, which could indicate a site's support for passkeys. Of course, these libraries use the same *Credentials* API that plain JS implemenations would, too. But the presence of *Hanko*'s JS code could be an indicator of higher confidence compared to a simple `navigator.credentials` call that could be part of a larger library like *SAP*'s *Gigya*, out of which, presumably, only a small part of the imported code is actually used on each site.

7.4.7. JavaScript Usage Detection

A significant amount of false positives could be prevented if there was a way to detect whether the part of a site's JS code containing a WebAuthn-related browser API call was ever used on the site or if it was part of *dead code* that is never executed. This could potentially exclude false positives like the 547 sites using *SAP*'s *Gigya* JS library. However, it is challenging to reliably detect whether some code in an unknown structure is ever executed. When combining all JS resources collected from a site and performing a static code analysis, it may be possible to identify code functions and branches that are never executed, though.

8. Conclusion

This thesis presented the first iteration of a distributed web crawler aimed at detecting the use of authentication methods to measure the adoption of phishing-resistant authentication on the web. It described architecting, implementing, and deploying the crawler and provided a comprehensive analysis of the resulting content corpus.

R1 posed the question of whether automatic authentication method detection was possible with reasonable accuracy. The results showed that it is indeed feasible to detect specific authentication methods through automatic content matching. However, the accuracy significantly varied across applied matching techniques. While DOM-based matching of HTML documents produced considerably fewer false positives compared to Regex-based matching of JS resources, the coverage rate is lower because WebAuthn-based MFA does not involve any HTML and sites supporting *passkeys*, i.e., discoverable multi-device FIDO credentials, are not required to use any HTML-based *Conditional UI*. The quantitative result validation showed that there is still potential for improvement in detection, as only 51.85 % of the sites labeled by a validation dataset ($N = 27$) were automatically identified to use some form of WebAuthn-based authentication. The qualitative analysis showed that the high number of false positives was mainly caused by a too lax URL discovery policy and third-party JS resources that included code using the WebAuthn API, which was not used on the visited sites.

With respect to R2, which asked whether the detection rate differed for static and dynamic web crawling, the results showed that a dynamic crawler, which can render client-side rendered content, is superior in collecting relevant content compared to traditional HTTP client-based content scraping. As the web is increasingly based on client-side rendered applications, crawling static content does not discover many JS resources that are available when intercepting real web browser traffic. In addition, the use case discriminates against static crawling since WebAuthn is a JS-based browser API.

In conclusion, this work has demonstrated that the automatic detection of phishing-resistant authentication on the web is possible. Even though certain aspects have the potential for optimization, the proposed method can help monitor the development towards more widespread adoption. A passwordless future is desirable because password-based authentication places an unnecessary burden on users and enables some of the most common attacks, like phishing and credential stuffing. With major technology corporations' recent endorsement of FIDO-based authentication, there is hope that passwords may soon be a thing of the past.

References

[1] 3rd Generation Partnership Project (3GPP). *Technical Realization of the Short Message Service (SMS)*. Technical Specification 23.040. 1999.

[2] Lawrence Abrams. "MFA Fatigue: Hackers' New Favorite Tactic in High-Profile Breaches". In: *BleepingComputer* (Sept. 20, 2022). URL: https://www.bleepingcomputer.com/news/security/mfa-fatigue-hackers-new-favorite-tactic-in-high-profile-breaches/ (visited on May 25, 2023).

[3] Yurij Ackermann. *Introduction to WebAuthn API*. Medium. Apr. 9, 2019. URL: https://medium.com/@herrjemand/introduction-to-webauthn-api-5fd1fb46c285 (visited on May 24, 2023).

[4] Yurij Ackermann. *Workshop: Authenticating Your Web Like a Boss*. 2018. URL: https://slides.com/fidoalliance/jan-2018-fido-seminar-webauthn-tutorial (visited on May 24, 2023).

[5] AgileBits Inc. *About Watchtower Privacy in 1Password*. Sept. 19, 2022. URL: https://support.1password.com/watchtower-privacy/ (visited on May 17, 2023).

[6] Furkan Alaca, AbdelRahman Abdou, and Paul C. van Oorschot. *Comparative Analysis and Framework Evaluating Mimicry-Resistant and Invisible Web Authentication Schemes*. Mar. 30, 2019. DOI: 10.48550/arXiv.1708.01706. preprint.

[7] Aftab Alam, Katharina Krombholz, and Sven Bugiel. "Poster: Let History Not Repeat Itself (This Time) – Tackling WebAuthn Developer Issues Early On". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London, UK: ACM, Nov. 6, 2019, pp. 2669–2671. DOI: 10.1145/3319535.3363283.

[8] Junade Ali. *Validating Leaked Passwords with K-Anonymity*. The Cloudflare Blog. Feb. 21, 2018. URL: https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/ (visited on Jan. 6, 2023).

[9] Amazon Web Services, Inc. *Amazon S3 REST API Introduction*. Amazon Simple Storage Service. Mar. 1, 2006. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html (visited on June 1, 2023).

[10] Amazon.com, Inc. *End of Service Notice*. Alexa.com. Nov. 19, 2022. URL: https://web.archive.org/web/20221119164239/https://www.alexa.com/login (visited on June 9, 2023).

[11] Olabode Anise and Kyle Lady. *State of the Auth*. White paper. Duo Security Inc., Nov. 7, 2017. URL: https://duo.com/assets/ebooks/state-of-the-auth.pdf (visited on Jan. 3, 2023).

[12] Apache Software Foundation. *Apache Cassandra*. URL: https://cassandra.apache.org (visited on June 9, 2023).

[13] Apache Software Foundation. *Compression*. Cassandra Documentation. Feb. 13, 2023. URL: https://cassandra.apache.org/doc/4.1/cassandra/operating/compression.html (visited on Apr. 13, 2023).

[14] Apple Inc. *Passkeys*. Apple Developer. URL: https://developer.apple.com/passkeys/ (visited on Jan. 5, 2023).

[15] Apple Inc. *Spotlight on: Passkeys*. Apple Developer. May 15, 2023. URL: https://developer.apple.com/news/?id=mgdnfp8w (visited on May 25, 2023).

[16] Apple Inc. *Supporting Passkeys*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/authenticationservices/public-private_key_authentication/supporting_passkeys (visited on Jan. 5, 2023).

[17] Ricardo Baeza-Yates and Carlos Castillo. "Crawling the Infinite Web". In: *Journal of Web Engineering* 6.1 (Oct. 30, 2007), pp. 049–072. ISSN: 1544-5976.

[18] Mehdi Bahrami, Mukesh Singhal, and Zixuan Zhuang. "A Cloud-Based Web Crawler Architecture". In: 18th International Conference on Intelligence in Next Generation Networks. Paris, France: IEEE, 2015, pp. 216–223. DOI: 10.1109/ICIN.2015.7073834.

[19] Luke Bakken et al. *Pika*. Version 1.3.2. Pika, May 5, 2023. URL: https://github.com/pika/pika (visited on June 13, 2023).

[20] Chris Bannister et al. *Gocql*. Version 1.3.2. Apr. 13, 2023. URL: https://github.com/gocql/gocql (visited on Apr. 13, 2023).

[21] Ben Baryo. *Restringer*. Version 1.7.1. June 6, 2023. URL: https://github.com/PerimeterX/restringer (visited on June 13, 2023).

[22] Felix Bause et al. *Hanko*. Version 0.6.0. Hanko GmbH, Apr. 19, 2023. URL: https://github.com/teamhanko/hanko (visited on May 24, 2023).
[23] Jorge Bay et al. *DataStax Node.Js Driver for Apache Cassandra*. Version 4.6.4. DataStax Inc., July 12, 2022. URL: https://github.com/datastax/nodejs-driver (visited on June 13, 2023).
[24] Vittorio Bertocci. *Our Take on Passkeys*. Auth0 Blog. Aug. 24, 2022. URL: https://auth0.com/blog/our-take-on-passkeys/ (visited on May 15, 2023).
[25] Haren Bhandari and Joe Viggiano. *The 2022 Web Almanac: CDN*. HTTP Archive, Oct. 13, 2022. URL: https://almanac.httparchive.org/en/2022/cdn (visited on Feb. 28, 2023).
[26] Eric Bidelman. *Getting Started with Headless Chrome*. Chrome Developers. Apr. 27, 2017. URL: https://developer.chrome.com/blog/headless-chrome/ (visited on June 16, 2023).
[27] Arnar Birgisson. *Security of Passkeys in the Google Password Manager*. Google Online Security Blog. Oct. 12, 2022. URL: https://security.googleblog.com/2022/10/SecurityofPasskeysintheGooglePasswordManager.html (visited on May 15, 2023).
[28] Bitdefender SRL. *Bitdefender Global Report: Cybersecurity and Online Behaviors*. 2021, p. 15. URL: https://www.bitdefender.com/files/News/CaseStudies/study/404/BD-Security-Behavior-Report-Final-at.pdf+ (visited on May 17, 2023).
[29] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692.
[30] Paolo Boldi et al. "UbiCrawler: A Scalable Fully Distributed Web Crawler". In: *Software: Practice and Experience* 34.8 (July 10, 2004), pp. 711–726. DOI: 10.1002/spe.587.
[31] John Bradley et al. *Client to Authenticator Protocol (CTAP)*. Review Draft v2.2. FIDO Alliance, Mar. 21, 2023.
[32] John Bradley et al. *Web Authentication: An API for Accessing Public Key Credentials*. W3C First Public Working Draft Level 3. W3C, Apr. 27, 2021.
[33] Christiaan Brand and Sriram Karra. *The Beginning of the End of the Password*. Google. May 3, 2023. URL: https://blog.google/technology/safety-security/the-beginning-of-the-end-of-the-password/ (visited on May 15, 2023).
[34] Christiaan Brand et al. *Client to Authenticator Protocol (CTAP)*. Implementation Draft v2.0. FIDO Alliance, Feb. 27, 2018.
[35] Brian Brazil et al. *JMX Exporter*. Version 0.18.0. The Linux Foundation, Mar. 7, 2023. URL: https://github.com/prometheus/jmx_exporter (visited on June 17, 2023).
[36] Michael Bridgen et al. *Amqplib*. Version 0.8.0. May 19, 2021. URL: https://github.com/amqp-node/amqplib (visited on June 13, 2023).
[37] Sergey Brin and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine". In: *Computer Networks and ISDN Systems* 30.1-7 (Apr. 1998), pp. 107–117. DOI: 10.1016/S0169-7552(98)00110-X.
[38] Matt Burgess. "Apple's Killing the Password. Here's Everything You Need to Know". In: *Wired* (Sept. 7, 2022). URL: https://www.wired.com/story/apple-passkeys-password-iphone-mac-ios16-ventura/ (visited on May 15, 2023).
[39] William E. Burr, Donna F. Dodson, and W. Timothy Polk. *Electronic Authentication Guideline*. NIST Special Publication 800-63v1.0.1. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2004. DOI: 10.6028/NIST.SP.800-63v1.0.1.
[40] Jeff Burt. "Multi-Factor Auth Fatigue Is Real - And It's Why You May Be in the Headlines Next". In: *The Register* (Mar. 11, 2022). URL: https://www.theregister.com/2022/11/03/mfa_fatigue_enterprise_threat/ (visited on May 25, 2023).
[41] Mathias Bynens and Peter Kvitek. *Chrome's Headless Mode Gets an Upgrade*. Chrome Developers. Feb. 22, 2023. URL: https://developer.chrome.com/articles/new-headless/ (visited on May 15, 2023).
[42] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems* 26.2 (June 2008), pp. 1–26. DOI: 10.1145/1365815.1365816.
[43] Kumar Chellapilla and Alexey Maykov. "A Taxonomy of JavaScript Redirection Spam". In: *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*. Banff, Alberta, Canada: ACM, May 8, 2007, pp. 81–88. DOI: 10.1145/1244408.1244423.
[44] Dave Childers. *State of the Auth*. Duo Labs Report. Cisco Systems, Inc., Sept. 14, 2021. URL: https://duo.com/assets/ebooks/state-of-the-auth-2021.pdf (visited on May 19, 2023).

[45] Junghoo Cho and Hector Garcia-Molina. “Parallel Crawlers”. In: *Proceedings of the 11th International Conference on World Wide Web*. Honolulu, Hawaii, USA: ACM, May 7, 2002, pp. 124–135. DOI: 10.1145/511446.511464.

[46] Cisco Systems Inc. *Duo Mobile*. Duo Security. URL: https://duo.com/product/multi-factor-authentication-mfa/duo-mobile-app (visited on May 25, 2023).

[47] Cloudflare Inc. *Bot Management & Protection*. Cloudflare. URL: https://www.cloudflare.com/products/bot-management/ (visited on May 31, 2023).

[48] Brian Coca et al. *Ansible*. Version 2.14.6. May 22, 2023. URL: https://github.com/ansible/ansible (visited on June 17, 2023).

[49] Yann Collet et al. *Lz4*. Version 1.9.4. Aug. 16, 2022. URL: https://github.com/lz4/lz4 (visited on June 19, 2023).

[50] Yann Collet et al. *Zstandard*. Version 1.5.5. Meta, Apr. 5, 2023. URL: https://github.com/facebook/zstd (visited on June 19, 2023).

[51] Barbara Collins. “Why Passkeys from Apple, Google, Microsoft May Soon Replace Your Passwords”. In: *CNBC* (Feb. 11, 2023). URL: https://www.cnbc.com/2023/02/11/why-apple-google-microsoft-passkey-should-replace-your-own-password.html (visited on May 15, 2023).

[52] Silvia Convento, Court Jacinic, and Becca Shareff. *Making Authentication Faster than Ever: Passkeys vs. Passwords*. Google Online Security Blog. May 5, 2023. URL: https://security.googleblog.com/2023/05/making-authentication-faster-than-ever.html (visited on May 15, 2023).

[53] DataStax Inc. *CQL Limits*. CQL for Cassandra 3.x. Feb. 18, 2022. URL: https://docs.datastax.com/en/cql-oss/3.x/cql/cql_reference/refLimits.html (visited on Apr. 20, 2023).

[54] Lucas Davi et al. “Over-the-Air Cross-Platform Infection for Breaking mTAN-based Online Banking Authentication”. In: Black Hat. Abu Dhabi, United Arab Emirates, Dec. 6, 2012.

[55] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[56] Alexis Deveria. *Web Authentication API*. Can I use... URL: https://caniuse.com/webauthn (visited on May 23, 2023).

[57] Docker Inc. *Enable IPv6 Support*. Docker Documentation. June 16, 2023. URL: https://docs.docker.com/config/daemon/ipv6/ (visited on June 17, 2023).

[58] Jason A. Donenfeld. “WireGuard: Next Generation Kernel Network Tunnel”. In: Network and Distributed System Security Symposium. San Diego, CA, USA, Feb. 27, 2017. DOI: 10.14722/ndss.2017.23160.

[59] Cristian Duda. “Searching Application Data”. PhD thesis. ETH Zurich, 2009. DOI: 10.3929/ETH-A-005844429.

[60] Cristian Duda et al. “AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications”. In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1440–1443. DOI: 10.14778/1454159.1454195.

[61] Jon Dugan et al. *iPerf*. URL: https://iperf.fr (visited on June 17, 2023).

[62] Jenny Edwards, Kevin McCurley, and John Tomlin. “An Adaptive Model for Optimizing Performance of an Incremental Web Crawler”. In: *Proceedings of the 10th International Conference on World Wide Web*. Hong Kong: ACM, Apr. 2001, pp. 106–113. DOI: 10.1145/371920.371960.

[63] James Elliott et al. *WebAuthn Library*. Duo Security, Dec. 5, 2022. URL: https://github.com/duo-labs/webauthn (visited on May 24, 2023).

[64] F-Secure Corporation. *Trojan: Android/Crusewind*. Threat Descriptions. June 2011. URL: https://www.f-secure.com/v-descs/trojan_android_crusewind.shtml (visited on May 18, 2023).

[65] Florian M Farke et al. “‘You Still Use the Password after All’ – Exploring FIDO2 Security Keys in a Small Company”. In: *Proceedings of the Sixteenth Symposium on Usable Privacy and Security*. USENIX Association, Aug. 2020, pp. 19–35. ISBN: 978-1-939133-16-8.

[66] Florian M Farke et al. “Exploring User Authentication with Windows Hello in a Small Business Environment”. In: *Proceedings of the Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 523–540. ISBN: 978-1-939133-30-4.

[67] Horst Feistel. “Cryptography and Computer Privacy”. In: *Scientific American* 228.5 (May 1973), pp. 15–23. DOI: 10.1038/scientificamerican0573-15.

[68] FIDO Alliance. *FIDO Alliance Member Companies & Organizations*. URL: https://fidoalliance.org/members/ (visited on May 18, 2023).

[69] FIDO Alliance. *Multi-Device FIDO Credentials*. White Paper. Mar. 2022. URL: https://fidoalliance.org/white-paper-multi-device-fido-credentials/ (visited on May 15, 2023).

[70] Lorenzo Franceschi-Bicchierai. "How to Protect Yourself From SIM Swapping Hacks". In: *Vice. Motherboard* (July 17, 2018). URL: https://www.vice.com/en/article/zm8a9y/how-to-protect-yourself-from-sim-swapping-hacks (visited on May 15, 2023).

[71] Vitaly Friedman. "Rethinking Authentication UX". In: *Smashing Magazine. General* (Aug. 4, 2022). URL: https://www.smashingmagazine.com/2022/08/authentication-ux-design-guidelines/ (visited on May 15, 2023).

[72] Nick Frymann et al. "Asynchronous Remote Key Generation: An Analysis of Yubico's Proposal for W3C WebAuthn". In: *Cryptology ePrint Archive, Paper 2020/1004* (Aug. 19, 2020). DOI: 10.1145/3372297.3417292.

[73] Michael Gilbert. *Package: Chromium - Disable/Swiftshader.Path*. Version 111.0.5563.110-1. URL: https://sources.debian.org/patches/chromium/111.0.5563.110-1/disable/swiftshader.patch/ (visited on Apr. 12, 2023).

[74] Lori Glavin. *Apple, Google and Microsoft Commit to Expanded Support for FIDO Standard to Accelerate Availability of Passwordless Sign-Ins*. FIDO Alliance. May 5, 2022. URL: https://fidoalliance.org/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard-to-accelerate-availability-of-passwordless-sign-ins/ (visited on Jan. 10, 2023).

[75] Lori Glavin. *Cloudflare Embraces FIDO to Help Its Own Security*. FIDO Alliance. Mar. 2, 2023. URL: https://fidoalliance.org/cloudflare-embraces-fido-to-help-its-own-security/ (visited on May 15, 2023).

[76] Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. "Weaponizing Femtocells: The Effect of Rogue Devices on Mobile Telecommunications". In: Network and Distributed System Security Symposium. San Diego, CA, USA, Feb. 6, 2012.

[77] Dan Goodin. "Google Passkeys Are a No-Brainer. You've Turned Them on, Right?" In: *Ars Technica* (May 8, 2023). URL: https://arstechnica.com/information-technology/2023/05/passwordless-google-accounts-are-easier-and-more-secure-than-passwords-heres-why/ (visited on May 15, 2023).

[78] Google LLC. *Chrome DevTools Protocol*. URL: https://chromedevtools.github.io/devtools-protocol/ (visited on June 9, 2023).

[79] Google LLC. *Chrome UX Report*. Chrome Developers. URL: https://developer.chrome.com/docs/crux/ (visited on June 9, 2023).

[80] Google LLC. *Key URI Format*. In collab. with Pier Fumagalli et al. URL: https://github.com/google/google-authenticator/wiki/Key-Uri-Format (visited on Jan. 5, 2023).

[81] Paul A Grassi et al. *Digital Identity Guidelines*. NIST Special Publication 800-63B. Gaithersburg, MD, USA: National Institute of Standards and Technology, Mar. 2, 2020. DOI: 10.6028/NIST.SP.800-63b.

[82] Thomas Habets et al. *Google/Google-Authenticator-Libpam*. Version 1.09. Google LLC, May 26, 2020. URL: https://github.com/google/google-authenticator-libpam (visited on May 19, 2023).

[83] Markus Hänel. *Puppeteer-Extra-Plugin-Stealth*. Version 2.11.2. Mar. 1, 2023. URL: https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth (visited on Apr. 12, 2023).

[84] Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. "Token Meets Wallet: Formalizing Privacy and Revocation for FIDO2". In: IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2023, pp. 978–995. DOI: 10.1109/SP46215.2023.00056.

[85] Christopher Harrell. *YubiKeys, Passkeys and the Future of Modern Authentication*. Yubico. Mar. 31, 2022. URL: https://www.yubico.com/blog/passkeys-and-the-future-of-modern-authentication/ (visited on Jan. 4, 2023).

[86] Mitchell Hashimoto et al. *Terraform*. Version 1.5.0. HashiCorp, June 12, 2023. URL: https://github.com/hashicorp/terraform (visited on June 17, 2023).

[87] Allan Heydon and Marc Najork. "Mercator: A Scalable, Extensible Web Crawler". In: *World Wide Web* 2.4 (Dec. 1999), pp. 219–229. DOI: 10.1023/A:1019213109274.

[88] Adam Holmberg et al. *DataStax Python Driver for Apache Cassandra*. Version 3.28.0. DataStax Inc., June 5, 2023. URL: https://github.com/datastax/python-driver (visited on June 13, 2023).

[89] Hang Hu et al. "Assessing Browser-level Defense against IDN-based Phishing". In: *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, Aug. 2021. ISBN: 978-1-939133-24-3.

[90] Troy Hunt. *I've Just Launched "Pwned Passwords" V2 With Half a Billion Passwords for Download*. Feb. 22, 2018. URL: https://www.troyhunt.com/ive-just-launched-pwned-passwords-version-2/ (visited on Jan. 8, 2023).

[91] Troy Hunt. *Open Source Pwned Passwords with FBI Feed and 225M New NCA Passwords Is Now Live!* Dec. 20, 2021. URL: https://www.troyhunt.com/open-source-pwned-passwords-with-fbi-feed-and-225m-new-nca-passwords-is-now-live/ (visited on May 17, 2023).

[92] IANA. *CBOR Object Signing and Encryption (COSE) Algorithm Registry*. Jan. 11, 2017. URL: https://www.iana.org/assignments/cose/cose.xhtml (visited on Mar. 12, 2023).

[93] Jamie Indigo and Dave Smart. *The 2022 Web Almanac: Page Weight*. HTTP Archive, Sept. 26, 2022. URL: https://almanac.httparchive.org/en/2022/page-weight (visited on Feb. 28, 2023).

[94] Internet Archive. *Digital Library of Free & Borrowable Books, Movies, Music & Wayback Machine*. URL: https://archive.org/ (visited on May 30, 2023).

[95] Internet Crime Complaint Center (IC3). *Internet Crime Report*. Federal Bureau of Investigation (FBI), 2022, p. 21. URL: https://www.ic3.gov/Media/PDF/AnnualReport/2022_IC3Report.pdf (visited on May 24, 2023).

[96] Internet Systems Consortium, Inc. *BIND 9*. URL: https://www.isc.org/bind/ (visited on June 17, 2023).

[97] Vincenzo Iozzo. *The Good, the Bad and the Ugly of Apple Passkeys*. SlashID Blog. Sept. 23, 2022. URL: https://www.slashid.dev/blog/passkeys-deepdive/ (visited on May 15, 2023).

[98] Vasu Jakkal. *This World Password Day Consider Ditching Passwords Altogether*. Microsoft Security Blog. May 5, 2022. URL: https://www.microsoft.com/en-us/security/blog/2022/05/05/this-world-password-day-consider-ditching-passwords-altogether/ (visited on Jan. 3, 2023).

[99] J. C. Jones and Tim Taubert. *Using Hardware Token-based 2FA with the WebAuthn API*. Mozilla Hacks – The Web Developer Blog. Jan. 16, 2018. URL: https://hacks.mozilla.org/2018/01/using-hardware-token-based-2fa-with-the-webauthn-api (visited on Jan. 16, 2023).

[100] Michael Jones et al. *Web Authentication: An API for Accessing Public Key Credentials*. Recommendation Level 1. W3C, Mar. 2019.

[101] Michael B. Jones, Akshay Kumar, and Emil Lundberg. *Web Authentication: An API for Accessing Public Key Credentials*. Editor's Draft Level 3. W3C, May 17, 2023.

[102] Brewster Kahle. *Let Us Serve You, but Don't Bring Us down*. Internet Archive Blogs. May 29, 2023. URL: https://blog.archive.org/2023/05/29/let-us-serve-you-but-dont-bring-us-down/ (visited on May 30, 2023).

[103] David Kargerl et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. El Paso, TX, USA: ACM Press, 1997, pp. 654–663.

[104] Markus Keil, Philipp Markert, and Markus Dürmuth. "'It's Just a Lot of Prerequisites': A User Perception and Usability Analysis of the German ID Card as a FIDO2 Authenticator". In: *EuroUSEC '22: Proceedings of the 2022 European Symposium on Usable Security*. Karlsruhe, Germany: ACM, Sept. 29, 2022, pp. 172–188. DOI: 10.1145/3549015.3554208.

[105] Anas Khan. *Soup*. Version v1.2.5. Jan. 16, 2022. URL: https://github.com/anaskhan96/soup (visited on June 13, 2023).

[106] Eiji Kitamura. *Passwordless Sign-in on Forms with WebAuthn Passkey Autofill*. Chrome Developers. Nov. 30, 2022. URL: https://developer.chrome.com/blog/webauthn-conditional-ui/ (visited on Jan. 12, 2023).

[107] Michael Klishin et al. *RabbitMQ Server*. Version 3.11.15. VMware Inc., Apr. 29, 2023. URL: https://github.com/rabbitmq/rabbitmq-server (visited on May 2, 2023).

[108] Thorin Klosowski. "RIP, Passwords. Here's What's Coming Next." In: *Wirecutter by The New York Times* (Jan. 11, 2023). URL: https://www.nytimes.com/wirecutter/blog/what-are-passkeys-and-how-they-can-replace-passwords/ (visited on May 15, 2023).

[109] Tobias Koppers et al. *Webpack*. Version 5.88.0. webpack, June 21, 2023. URL: https://github.com/webpack/webpack (visited on June 26, 2023).

[110] Mikhail Korobov et al. *Splash*. Version 3.5.0. June 16, 2020. URL: https://github.com/scrapinghub/splash (visited on June 1, 2023).

[111] Martijn Koster et al. *Robots Exclusion Protocol*. Request for Comments 9309. RFC Editor, Sept. 2022.

[112] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. Request for Comments 2104. RFC Editor, Feb. 1997.

[113] Brian Krebs. "Your Phone May Soon Replace Many of Your Passwords". In: *Krebs on Security* (May 9, 2022). URL: https://krebsonsecurity.com/2022/05/your-phone-may-soon-replace-many-of-your-passwords/ (visited on May 15, 2023).

[114] Adam Langley. *Passkeys*. ImperialViolet. Sept. 22, 2022. URL: https://www.imperialviolet.org/2022/09/22/passkeys.html (visited on May 15, 2023).

[115] Leona Lassak et al. "It's Stored, Hopefully, on an Encrypted Server: Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn". In: *Proceedings of the 30th USENIX Security Symposium*. Aug. 2021, pp. 91–108. ISBN: 978-1-939133-24-3.

[116] Victor Le Pochat et al. *Tranco*. URL: https://tranco-list.eu/ (visited on June 9, 2023).

[117] Victor Le Pochat et al. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society, 2019. DOI: 10.14722/ndss.2019.23386.

[118] Noah Levitt et al. *Heritrix*. Version 3.4.0. Internet Archive, July 27, 2022. URL: https://github.com/internetarchive/heritrix3 (visited on Feb. 17, 2023).

[119] Boon Thau Loo, Owen Cooper, and Sailesh Krishnamurthy. *Distributed Web Crawling over DHTs*. University of California, Berkeley Department of Electrical Engineering and Computer Sciences Technical Report CSD-04-130. Feb. 2004.

[120] Abraão Lourenço. *Supporting Passkeys in Shop's Authentication Flows*. Shopify Engineering. Mar. 24, 2023. URL: https://shopify.engineering/supporting-passkeys-in-shop-authentication-flows (visited on May 15, 2023).

[121] Emil Lundberg. *Yubico Proposes WebAuthn Protocol Extension to Simplify Backup Security Keys*. Yubico. Nov. 16, 2020. URL: https://www.yubico.com/blog/yubico-proposes-webauthn-protocol-extension-to-simplify-backup-security-keys/ (visited on Jan. 20, 2023).

[122] Andrey Lushnikov et al. *Puppeteer*. Version 20.6.0. June 9, 2023. URL: https://github.com/puppeteer/puppeteer (visited on June 9, 2023).

[123] Nathan Lutchansky. *TAYGA - NAT64 for Linux*. Version 0.9.2. June 10, 2011. URL: http://www.litech.org/tayga/ (visited on June 17, 2023).

[124] Sanam Ghorbani Lyastani et al. "Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication". In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 842–859. DOI: 10.1109/SP40000.2020.00047.

[125] David M'Raihi et al. *HOTP: An HMAC-Based One-Time Password Algorithm*. Request for Comments 4226. RFC Editor, Dec. 2005.

[126] David M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. Request for Comments 6238. RFC Editor, May 2011.

[127] Majestic-12 Ltd. *Majestic Million*. URL: https://majestic.com/reports/majestic-million (visited on Feb. 23, 2023).

[128] Jim Manico et al. *Multifactor Authentication Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html (visited on May 18, 2023).

[129] Daniel Martí et al. *Chromedp*. Version 0.9.1. Apr. 15, 2023. URL: https://github.com/chromedp/chromedp (visited on June 13, 2023).

[130] Celso Martinho and Sabina Zejnilovic. *Goodbye, Alexa. Hello, Cloudflare Radar Domain Rankings*. The Cloudflare Blog. Sept. 30, 2022. URL: http://blog.cloudflare.com/radar-domain-rankings/ (visited on May 15, 2023).

[131] MDN Contributors. *PublicKeyCredential: isConditionalMediationAvailable() Static Method*. MDN Web Docs. June 12, 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/PublicKeyCredential/isConditionalMediationAvailable (visited on June 13, 2023).

[132] Jens Oliver Meiert. *The 2022 Web Almanac: Markup*. 3. HTTP Archive, Sept. 26, 2022.

[133] Nick Mooney, Nick Steele, and Jeremy Erickson. *Network Transport Summary*. Duo Security, May 4, 2020.

[134] Neal Mueller. *Credential Stuffing*. In collab. with Jmanico et al. OWASP Foundation. URL: https://owasp.org/www-community/attacks/Credential_stuffing (visited on May 17, 2023).

[135] Sebastian Nagel et al. *Apache Nutch*. Version 1.19. The Apache Software Foundation, Aug. 22, 2022. URL: https://github.com/apache/nutch (visited on Feb. 17, 2023).

[136] Marc Najork. "Web Crawler Architecture". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2017. DOI: 10.1007/978-1-4899-7993-3_457-3.

[137] Marc Najork and Allan Heydon. "High-Performance Web Crawling". In: *Handbook of Massive Data Sets*. Ed. by James Abello, Panos M. Pardalos, and Mauricio G. C. Resende. Vol. 4. Boston, MA, USA: Springer, 2002, pp. 25–45. DOI: 10.1007/978-1-4615-0005-6_2.

[138] Dain Nilsson. *Yubico's Take on U2F Key Wrapping*. Yubico. Nov. 14, 2014. URL: https://www.yubico.com/blog/yubicos-u2f-key-wrapping/ (visited on May 23, 2023).

[139] Karsten Nohl and Chris Paget. "GSM: SRSLY?" 26th Chaos Communication Congress (Berlin). Jan. 9, 2010. URL: https://fahrplan.events.ccc.de/congress/2009/Fahrplan/events/3654.en.html (visited on Jan. 24, 2023).

[140] Torkel Ödegaard et al. *Grafana*. Version 9.5.3. Grafana Labs, June 6, 2023. URL: https://github.com/grafana/grafana (visited on June 17, 2023).

[141] Okta, Inc. *Set up Okta Verify on iOS Devices*. Okta Help Center. May 1, 2023. URL: https://help.okta.com/eu/en-us/Content/Topics/end-user/ov-setup-ios.htm (visited on May 25, 2023).

[142] Christopher Olston and Marc Najork. "Web Crawling". In: *Foundations and Trends® in Information Retrieval* 4.3 (2010), pp. 175–246. DOI: 10.1561/1500000017.

[143] OWASP Foundation. *Credential Stuffing Prevention*. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html (visited on May 17, 2023).

[144] Addison Phillips and Mark Davis. *Tags for Identifying Languages*. Request for Comments 5646. RFC Editor, Sept. 2009.

[145] David Pierce. "Dashlane Is Ready to Replace All Your Passwords with Passkeys". In: *The Verge* (Aug. 31, 2022). URL: https://www.theverge.com/2022/8/31/23329373/dashlane-passkeys-password-manager (visited on May 25, 2023).

[146] Brian Pinkerton. "WebCrawler: Finding What People Want". PhD thesis. Seattle, Washington, USA: University of Washington, 2000.

[147] Ansuman Prusty et al. "Horizontally Scalable Web Crawler Using Containerization and a Graphical User Interface". In: *International Journal of Engineering Research and* 09.05 (May 15, 2020). DOI: 10.17577/IJERTV9IS050268.

[148] Q-Success DI Gelbmann GmbH. *Historical Yearly Trends in the Usage Statistics of Javascript Libraries for Websites*. Web Technology Surveys (W3Techs). June 2023. URL: https://w3techs.com/technologies/history_overview/javascript_library/all/y (visited on June 1, 2023).

[149] Do Le Quoc et al. "UniCrawl: A Practical Geographically Distributed Web Crawler". In: 8th International Conference on Cloud Computing (CLOUD). New York City, NY, USA: IEEE, June 2015, pp. 389–396. DOI: 10.1109/CLOUD.2015.59.

[150] Björn Rabenstein et al. *Prometheus Go Client Library*. Version 1.15.0. The Linux Foundation, Apr. 13, 2023. URL: https://github.com/prometheus/client_golang (visited on June 17, 2023).

[151] Suby Raman. *Guide to Web Authentication*. Duo Security. URL: https://webauthn.guide (visited on May 23, 2023).

[152] Rishu Ranjan. *Password Spraying Attack*. OWASP Foundation. URL: https://owasp.org/www-community/attacks/Password_Spraying_Attack (visited on May 17, 2023).

[153] Fabian Reinartz et al. *Prometheus*. Version 2.44.0. The Linux Foundation, May 13, 2023. URL: https://github.com/prometheus/prometheus (visited on June 17, 2023).

[154] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser". In: *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA, USA, Aug. 2019. ISBN: 978-1-939133-06-9.

[155] Charlie Reis. *Multi-Process Architecture*. Chromium Blog. Sept. 11, 2008. URL: https://blog.chromium.org/2008/09/multi-process-architecture.html (visited on June 15, 2023).

[156] Leonard Richardson. *Beautifulsoup4*. Version 4.12.2. Apr. 7, 2023. URL: https://www.crummy.com/software/BeautifulSoup/bs4/ (visited on June 13, 2023).

[157] River Bank Computing Ltd. *PyQt5: Python Bindings for the Qt Cross Platform Application Toolkit*. Version 5.14.0. Dec. 19, 2020. URL: https://www.riverbankcomputing.com/software/pyqt/ (visited on June 1, 2023).

[158] Eric Rubin. *How We Boosted WebAuthn Adoption from 20 Percent to 93 Percent in Two Days*. GitLab. Nov. 9, 2022. URL: https://about.gitlab.com/blog/2022/11/09/how-we-boosted-webauthn-adoption-from-20-percent-to-93-percent-in-2-days/ (visited on May 15, 2023).

[159] Trevor Russo. *SIMulated Trust: How Malicious Actors Take Advantage of Cellular Carriersto Perform SIM Swapping Attacks*. Boston, MA, USA: Tufts University, Dec. 13, 2019. URL: https://www.cs.tufts.edu/comp/116/archive/fall2019/trusso.pdf (visited on Feb. 15, 2023).

[160] Kimberly Ruth et al. "Toppling Top Lists: Evaluating the Accuracy of Popular Website Lists". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. Nice, France: ACM, Oct. 25, 2022, pp. 374–387. DOI: 10.1145/3517745.3561444.

[161] Pierangela Samarati and Latanya Sweeney. "Protecting Privacy When Disclosing Information: K-Anonymity and Its Enforcement through Generalization and Suppression". In: *Proceedings of the IEEE Symposium on Research in Security and Privacy (S&P)*. Oakland, CA, USA: IEEE, May 1998.

[162] Salvatore Sanfilippo et al. *Redis*. Version 7.0.11. Apr. 17, 2023. URL: https://github.com/redis/redis (visited on June 1, 2023).

[163] Evan Sangeline. *It Is *not* Possible to Detect and Block Chrome Headless*. Intoli Blog. Jan. 18, 2018. URL: https://intoli.com/blog/not-possible-to-block-chrome-headless/ (visited on Apr. 11, 2023).

[164] Evan Sangeline. *Making Chrome Headless Undetectable*. Intoli Blog. Aug. 9, 2017. URL: https://intoli.com/blog/making-chrome-headless-undetectable/ (visited on Apr. 11, 2023).

[165] SAP. *What Is Gigya*. URL: https://www.sap.com/products/acquired-brands/what-is-gigya.html (visited on June 23, 2023).

[166] Nina Satragno and Jeff Hodges. *Explainer: WebAuthn Conditional UI*. W3C WebAuthn Wiki. Oct. 5, 2022. URL: https://github.com/w3c/webauthn (visited on Dec. 5, 2022).

[167] Jim Schaad. *CBOR Object Signing and Encryption (COSE)*. Request for Comments 8152. RFC Editor, July 2017.

[168] Peter Schartner and Stefan Burger. *Attacking mTAN-Applications like e-Banking and Mobile Signatures*. Technical Report TR-syssec-11-01. University of Klagenfurt, Dec. 2011.

[169] Uri Schonfeld and Narayanan Shivakumar. "Sitemaps: Above and beyond the Crawl of Duty". In: *Proceedings of the 18th International Conference on World Wide Web*. Madrid, Spain: ACM, Apr. 20, 2009, pp. 991–1000. DOI: 10.1145/1526709.1526842.

[170] Fabian Schwarz, Khue Do, and Gunnar Heide. "FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles, CA, USA: ACM, Nov. 2022, pp. 2581–2594. DOI: 10.1145/3548606.3560584.

[171] Claude E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27.4 (1948), pp. 623–656.

[172] Hossein Siadati et al. "Mind Your SMSes: Mitigating Social Engineering in Second Factor Authentication". In: *Computers & Security* 65 (Mar. 2017), pp. 14–28. DOI: 10.1016/j.cose.2016.09.009.

[173] Software Freedom Conservancy. *Selenium*. URL: https://www.selenium.dev/ (visited on June 1, 2023).

[174] Lance Spitzner. *Time for Password Expiration to Die*. SANS Cyber Security Blog. June 27, 2019. URL: https://www.sans.org/blog/time-for-password-expiration-to-die/ (visited on May 22, 2023).

[175] Tristan Tarrant et al. *Infinispan, In-Memory Distributed Data Store*. Version 14.0.6. Red Hat, Jan. 19, 2023. URL: https://github.com/infinispan/infinispan (visited on Feb. 17, 2023).

[176] Kyle Taylor and Laura Silver. *Smartphone Ownership Is Growing Rapidly Around the World, but Not Always Equally*. Pew Research Center, Feb. 5, 2019. URL: https://www.pewresearch.org/global/wp-content/uploads/sites/2/2019/02/Pew-Research-Center_Global-Technology-Use-2018_2019-02-05.pdf (visited on Apr. 24, 2023).

[177] The Apache Software Foundation. *Frequently Asked Questions*. Apache Cassandra Documentation. URL: https://cassandra.apache.org/doc/4.0/cassandra/faq/#can-large-blob (visited on Apr. 20, 2023).

[178] The Open Group. "General Concepts". In: *Base Specifications*. Issue 7. IEEE, 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html (visited on July 6, 2020).

[179] Anna Tims. "'Sim Swap' Gives Fraudsters Access-All-Areas via Your Mobile Phone". In: *The Guardian* (Sept. 26, 2015). URL: https://www.theguardian.com/money/2015/sep/26/sim-swap-fraud-mobile-phone-vodafone-customer (visited on May 18, 2023).

[180] Mindy Tran, Sabrina Amft, and Dominik Wermke. "Poster: User Awareness of Phishing and WebAuthn". In: 43rd IEEE Symposium on Security and Privacy, IEEE S&P 2022. San Francisco, CA, USA, May 2022. URL: https://www.ieee-security.org/TC/SP2022/downloads/SP22-posters/sp22-posters-53.pdf.

[181] Sean Treadway et al. *Amqp091-Go*. Version 1.8.1. RabbitMQ, May 4, 2023. URL: https://github.com/rabbitmq/amqp091-go (visited on June 13, 2023).

[182] Uber Technologies Inc. *Security Update*. Uber Newsroom. Sept. 16, 2022. URL: https://www.uber.com/newsroom/security-update (visited on May 25, 2023).

[183] Enis Ulqinaku et al. "Is Real-time Phishing Eliminated with FIDO?" In: *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, Aug. 2021, pp. 3811–3828. ISBN: 978-1-939133-24-3.

[184] unix-ninja. *Attacking Google Authenticator*. Oct. 22, 2018. URL: https://www.unix-ninja.com/p/attacking_google_authenticator (visited on May 19, 2023).

[185] Sander Vinberg and Jarrod Overson. *Credential Stuffing Report*. F5 Labs, Feb. 9, 2021. URL: https://www.f5.com/labs/articles/threat-intelligence/2021-credential-stuffing-report (visited on May 17, 2023).

[186] VMware Inc. *Networking and RabbitMQ*. RabbitMQ Documentation. Jan. 2, 2023. URL: https://www.rabbitmq.com/networking.html (visited on Apr. 12, 2023).

[187] Jeremy Wagner and Barry Pollard. *Time to First Byte (TTFB)*. web.dev. Jan. 19, 2023. URL: https://web.dev/ttfb/ (visited on June 6, 2023).

[188] Michael Webster. *How Passkeys Reduce Friction in the Ecommerce Shopping Experience*. Shopify. Feb. 9, 2023. URL: https://www.shopify.com/blog/ecommerce-payment-authentication (visited on May 15, 2023).

[189] Alex Weinert. *Your Pa$$word Doesn't Matter*. Azure Active Directory Identity Blog. July 9, 2019. URL: https://techcommunity.microsoft.com/t5/azure-active-directory-identity/your-pa-word-doesn-t-matter/ba-p/731984 (visited on May 23, 2023).

[190] Steve Won. *Goodbye, Passwords*. 1Password Blog. Feb. 9, 2023. URL: https://blog.1password.com/unlock-1password-with-passkeys/ (visited on May 18, 2023).

[191] Yahoo Japan. *Yahoo! JAPAN Is No Longer Available in the EEA and the United Kingdom*. Apr. 6, 2022. URL: https://www.yahoo.co.jp/ (visited on June 23, 2023).

[192] YouGov PLC. *Passwords*. Online Survey. Oct. 2018. URL: https://d25d2506sfb94s.cloudfront.net/cumulus_uploads/document/81iu1qrr2x/Passwords%20results,%20Sept.%202017%20%E2%80%93%20Oct.%202018.pdf (visited on May 19, 2023).

[193] Diego Zavala et al. *Bringing Passkeys to Android & Chrome*. Android Developers Blog. Oct. 12, 2022. URL: https://android-developers.googleblog.com/2022/10/bringing-passkeys-to-android-and-chrome.html (visited on Jan. 3, 2023).

A. Appendix

Listing 13: Regular Expressions for Authentication URL Matching

```
log-?in(\W|$)
auth(enticate)?(\W|$)
register(\W|$)
registration(\W|$)
account(\W|$)
sign-?(in|up)(\W|$)
admin(\W|$)
```