Requirements Analysis, System Architecture and Evaluation of a Privacy-First Web Application Framework

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Stefan Gussner
Registration Number 01527253

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Advisor: Thomas Grechenig

Wien, 28th August, 2024

Declaration of Authorship

Stefan Gussner

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed. I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, 28th August, 2024

Stefan Gussner

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die zum Gelingen dieser Diplomarbeit beigetragen haben.

Mein besonderer Dank gilt Thomas Grechenig für die Betreuung meiner Arbeit. Ich schätze die Möglichkeit, unter seiner Anleitung an diesem Thema arbeiten zu dürfen.

Ein herzlicher Dank gilt auch Clement Hlauschek, der mir als Kontaktperson wertvolle Unterstützung und konstruktives Feedback gegeben hat. Seine Anregungen und seine Zeit haben maßgeblich zur Fertigstellung dieser Arbeit beigetragen.

Des Weiteren möchte ich meiner Familie und meinen Freunden danken, die mich während des gesamten Studiums unterstützt haben.

Vielen Dank!

Acknowledgements

At this point, I would like to thank everyone who contributed to the success of this thesis.

My special thanks go to Thomas Grechenig for supervising my work. I appreciate the opportunity to work on this topic under his guidance.

I would also like to thank Clement Hlauschek, who provided valuable support and constructive feedback as a contact person. His suggestions and time have significantly contributed to the completion of this work.

Furthermore, I would like to thank my family and friends who supported me throughout my studies.

Thank you!

Kurzfassung

Aus ökonomischen Gründen ist Software in den letzten Jahren zentralisierter geworden. Webanwendungen bieten eine bequeme Möglichkeit für Benutzer, eine Anwendung zu starten, ohne Software installieren zu müssen. Diese Bequemlichkeit geht jedoch bei traditionellen webarchitekturen auf Kosten der Privatsphäre. In dieser Arbeit schlagen wir ein privacy-first Webanwendungsframework vor, das seine Abhängigkeit von zentralisierter Infrastruktur minimiert. Wir präsentieren eine Anforderungsanalyse, die Systemarchitektur und eine Bewertung des Frameworks anhand von drei Beispielanwendungen. Wir zeigen, dass das Framework die Entwicklung von Peer-to-Peer-Anwendungen ermöglicht, die resistent gegen Zensur sind. Es gibt jedoch Einschränkungen in Browser-APIs und der Verfügbarkeit von Bibliotheken für die Webplattform, die die Anwendungen, die derzeit mit dem Framework entwickelt werden können, einschränken.

Keywords: *privacy, web application, peer-to-peer, CRDT, censorship resistance*

Abstract

Due to economic reasons, software has grown more centralized in recent years. Web applications offer a convenient way for users to start using an application without the need to install software. However, this convenience comes at the cost of privacy. In this work we propose a privacy-first web application framework that minimizes its dependency on centralized infrastructure. We present a requirements analysis, the system architecture and an evaluation of the framework based on three sample applications. We demonstrate that the framework allows for the development of peer-to-peer applications that are resistant to censorship. However, there are restrictions in web browser APIs and availability of libraries for the web platform that restrict the applications that can currently be developed with the framework.

Keywords: *privacy, web application, peer-to-peer, CRDT, censorship resistance*

Contents

CHAPTER 1 Introduction

In this chapter, we introduce the topic of this work, describe the problem, and outline the goals.

1.1 Problem Description

In recent years, there is a push for centralization in the software development world. Applications need to be scalable to billions of users to collect user data for targeted advertising to be profitable. To reverse this trend and protect the privacy of individuals, the European Union created the General Data Protection Regulation (GDPR) [1]. GDPR requires that the processing of personal data shall be “limited to what is necessary in relation to the purposes for which they are processed (‘data minimization’)” and “processed in a manner that ensures appropriate security of the personal data, including protection against unauthorized or unlawful processing and against accidental loss, destruction, or damage, using appropriate technical or organizational measures (‘integrity and confidentiality’)”.

The traditional web architecture consists of a back-end with an application server processing plain-text information and storing data centralized and unencrypted in one or multiple databases [2], [3], [4]. Compromising a database reveals user data to an attacker.

Attacks on centralized and unencrypted database systems are not restricted to those outside of the organization running such systems. Insider attacks, where employees leak or abuse data, are also a threat. In 2019, insiders leaked 153 audio recordings captured by the “Google Assistant” smart speaker system to a news outlet[1].

Inside attacks are sometimes mandated by national governments. In 2021, a man sent pictures of his naked son to a doctor for diagnosis and is reported to the police for

[1]`https://siliconangle.com/2019/07/11/privacy-fail-audio-recorded-google-assistant-leaked-belgian-news-outlet/` (last visited 2024-08-28)

suspected child abuse by Google. Android phones can easily be set up to automatically upload all pictures taken to the Google Photos cloud, and Google uses a neural network to detect pictures of naked children among all uploaded pictures. Even after the police complete the investigation and found no violation of the law, the man did not get his Google account, which contains all his email, contacts, and photos, back[2]. Data commonly collected by web applications can be exploited in various ways, such as price discrimination, spam, and identity theft [5].

Even though centralized data collection claims to anonymize their records, Deußer et al. [6] argue that while it may be possible to store single coarsened clicks anonymously, any collection of higher complexity will contain large amounts of pseudonymous data.

Among others, Fuller et al. [7], Lewi et al. [8], and Wagner et al. [9] propose encrypted databases as part of a solution for privacy concerns in centralized applications. However, Poddar et al. [10] have demonstrated that applications using encrypted databases can still leak information to an attacker. Centralized systems allow for a wide range of data to be collected, including metadata such as the time a client is active. Centrally stored data is inherently more attractive to attackers as it allows the compromise of many accounts at once rather than having to perform multiple individual attacks.

1.2 Goals

To circumvent the privacy and security problems of unencrypted centralized data storage for web applications processing sensitive data, this work proposes a JavaScript framework named “Applink” offering an interface for decentralized applications for secure persistent data storage and partial transactional state replication. Applink should support latest version of the three major web browsers Google Chrome[3], Mozilla Firefox[4], and Apple Safari[5] from now on called “current web browsers”. Applink sends messages that are encrypted and authenticated between instances of the web application. To reduce enrty barriers for users[6], Applink does not rely on web browser extensions and is transparent to the users. This work will not implement the full functionality of Applink but will instead focus on prototyping key parts of the framework to determine the feasibility of the approach.

Such a framework poses the following research questions:

The investigation (1) evaluates the possibility of implementing the Applink JavaScript framework with the APIs provided by current web browsers without extensions. This is especially important for web browsers on mobile devices, which often cannot install

[2]`https://www.nytimes.com/2022/08/21/technology/google-surveillance-toddler-photo.html` (last visited 2024-08-28)

[3]`https://www.google.com/chrome/` (last visited 2024-08-28)

[4]`https://www.mozilla.org/firefox/` (last visited 2024-08-28)

[5]`https://www.apple.com/safari/` (last visited 2024-08-28)

[6]“users” always refers to the users of a web application; “developers” always refers to the developers using Applink to develop a web application

extensions. This work (2) explores and documents methods to ensure data consistency in a partially replicated system within the constraints of a web application. The use of hardware security tokens for authentication has been standardized by the W3C[7]. This work (3) explores the possibility of using hardware security tokens and other mechanisms in web browsers to encrypt application data in a user-friendly way without relying on passwords. Last but not least, the investigation (4) compares the performance of Applink to the performance of centralized web applications.

In this chapter, we have established that the current architecture of web applications poses a privacy risk to users. We have also outlined the goals, which aim to develop a JavaScript framework that provides the basis for decentralized web applications. In the next chapter, we will discuss the methodological approach to achieving these goals.

[7]`https://www.w3.org/TR/2021/REC-webauthn-2-20210408/`(last visited 2024-08-28)

CHAPTER 2

Methodological Approach

The methodological approach [11] consists of the following eight steps:

1. The first step is to conduct a review of the academically published literature. The focus of the literature review is on distributed database systems, in-browser encryption, and message exchange methods. This step includes analyzing the reasons why browser extensions have been used in the past

2. The second step is to conduct a use-case analysis of three sample applications. This step uses three different applications to evaluate the generalizability of the proposed approach.

   2.1 The first sample application is a beer credit system (BCS). The BCS allows users to pay for beer in advance and keep track of remaining credit. Administrators must approve deposits. Users can purchase beer using credits at any time without approval. The implementation of the BCS demonstrates how administrative functionality can be implemented in Applink.

   2.2 The second sample application is an invoicing system (IS). The IS allows users to receive invoices from other applications. Users can view the invoices and see statistics about their purchases. The implementation of the IS demonstrates how to send data between different applications (Cross-Origin Resource Sharing (CORS)[1]) using Applink. This ability is not only important for application interoperability but also for censorship resistance, as applications could transfer user data to the same application running on a new domain to circumvent a domain takedown. This is possible because the web application can outlive a takedown for multiple days using the Service Worker API[2].

[1] `https://www.w3.org/TR/2020/SPSD-cors-20200602/` (last visited 2024-08-28)
[2] `https://www.w3.org/TR/service-workers/` (last visited 2024-08-28)

   2.3 The third sample application is a note app. The note app allows one or multiple users to create and edit text notes. The implementation of the note app demonstrates how collaborative editing can be implemented using Applink.

3. The third step is to derive the requirements (including a threat model) for Applink based on the use-case analysis.

4. The fourth step is to explore techniques and APIs to implement Applink based on the requirements derived in the previous step. The exploration is performed by implementing key parts of the prototype framework, such as communication and data storage, using multiple approaches and comparing the performance and reliability of the approaches.

5. The fifth step is to use the techniques and APIs found in the previous step to implement Applink to the extent possible in current web browsers. Any developer familiar with web technologies should be able to use Applink.

6. The sixth step is to document the pain points and limitations of implementing such a framework. This occurs in parallel with the development of the prototype framework.

7. The seventh step is to implement the sample applications using the prototype framework.

8. The eighth step is to evaluate the prototype framework in terms of performance, end-user experience, security, and privacy. This step includes testing if the prototype applications meet the requirements derived from the use-case analysis.

   8.1 The performance analysis is conducted using the developer tools of a current web browser. The three sample applications are evaluated for their adherence to the goals stated in the RAIL model[3] based on the work of Nielsen et al. [12]. The RAIL model limits the response time to user input to under 100ms, time to draw a frame to under 16ms, time to load the page to under 5000ms, and idle processing time to under 50ms. The performance analysis is performed using Google Chrome[4] on a Google Pixel 6 Pro running the Android Operating System.

   8.2 The end-user experience is argued based on the requirements described in step three.

   8.3 The security of Applink is systematically (but without formal proof) analyzed and argued based on the threat model and the security requirements described in step three.

[3]`https://web.dev/rail/` (last visited 2024-08-28)
[4]`https://www.google.com/chrome/` (last visited 2024-08-28)

8.4 The privacy of Applink is systematically (but without formal proof) analyzed and argued based on the threat model and the privacy requirements described in step three. The analysis of the privacy of Applink includes a statistical analysis of the timings and packet sizes of the messages exchanged. Furthermore, the privacy analysis includes reasoning about the possibility of an adversary being able to reconstruct the state of an application based on observing the exchanged messages.

In this chapter, we have outlined the methodological approach to achieve the goals of this work. In the next chapter, we will discuss related work.

CHAPTER 3

Related Work

In this chapter, we outline relevant related work, focusing on two main areas: privacy-enhancing technologies for web applications and various attacks on system components relevant to Applink.

3.1 DHT / Overlay Networks

Because Applink is a decentralized web application framework, it requires a decentralized network to facilitate communication between users. In this section, we discuss related work on overlay network technologies and potential attack vectors targeting these networks.

Stoica et al. [13] describe the Chord protocol, a distributed hash table (DHT)-based overlay network that provides a scalable and efficient method for locating data in a decentralized network. DHT systems are widely used in peer-to-peer overlay networks to facilitate efficient data retrieval and storage.

DHTs form the foundation of many decentralized systems. One such system is the InterPlanetary File System (IPFS)[1], a peer-to-peer data storage and retrieval system. Prünster et al. [14] describe an eclipse attack on the IPFS network by precomputing Sybil nodes and using them to eclipse a target node. They identify a flaw in the DHT peer scoring mechanism that awards points for unsolicited advertisement of content results and other non-supportive behavior, highlighting the need for improved peer scoring mechanisms to prevent such attacks. Prünster et al.'s work underscores the importance of robust peer scoring mechanisms in DHT-based systems to prevent eclipse attacks.

Another approach to storing data in a decentralized fashion is used by the Bitcoin network. The Bitcoin network employs a gossip protocol because every full node must know about all transactions that have occurred on the network. Heilman et al. [15] describe an

[1]https://ipfs.tech/ (last visited 2024-08-28)

eclipse attack on the Bitcoin network, where an attacker controls all connections to a node in a peer-to-peer network. This control enables the attacker to carry out a double spend attack with less than 50% of mining power. They propose several countermeasures, including evicting peers based on hash collisions of their IP addresses, testing known peers and replacing them only if they are offline, and using Tor-style guard nodes to enhance security.

Tran et al. [16] describe a stealthier eclipse attack compared to that of Heilman et al. on the Bitcoin network that can be performed by a more powerful adversary capable of controlling (or spoofing) a significant portion of the network. This type of attack is feasible only for state-sponsored network adversaries, emphasizing the threat posed by attackers with substantial resources.

Bienstock et al. [17] propose the ASMesh protocol, an augmentation of the Signal protocol tailored for the unique characteristics of mesh networking. They introduce a message anonymizer that symmetrically encrypts double ratchet encrypted packets using some of the established key material from the ratchet and a key derivation function (KDF), enhancing privacy and security in mesh networks.

To encrypt messages in a decentralized network, developers can use the libsignal library[2]. The Signal protocol is used in popular messaging applications such as WhatsApp[3,4] and Signal[5]. Albrecht et al. [18] reveal that using libsignal, the library at the core of the privacy-focused Signal chat application, alone does not guarantee confidentiality. Their investigation into the offline messaging application Bridgefy identified a security vulnerability in the application's compress-then-encrypt schema, which can be exploited by analyzing ciphertext lengths within a small set of possible plaintexts. Additionally, they note that an adversary can use publicly visible user IDs to build communication graphs. Our work attempts to address the issue of publicly visible user IDs, aiming to enhance privacy and security.

In recent years, neural networks have become a popular tool for providing users with convenience functions, such as the autocomplete functionality of the Google Keyboard, as described by Yang et al. [19]. These neural networks can be trained in a decentralized manner using federated learning. However, this approach is only effective if the participants are honest, as it can leak information about the training data.

Wink et al. [20] describe a method for the distributed training of neural networks using a secure average computation to summarize model weights. Their method protects against model reverse engineering but does not safeguard against dishonest participants who submit incorrect model weights.

[2] `https://github.com/signalapp/libsignal` (last visited 2024-08-28)
[3] `https://signal.org/blog/whatsapp-complete/` (last visited 2024-08-28)
[4] `https://faq.whatsapp.com/820124435853543/?helpref=hc_fnav` (last visited 2024-08-28)
[5] `https://signal.org/` (last visited 2024-08-28)

Luqman et al. [21] demonstrate that distributed training of neural networks is vulnerable to membership inference attacks due to the tendency of neural networks to memorize data.

This thesis uses a DHT-based overlay mesh network to facilitate communication between users. The related work in the area outlines potential pitfalls and vulnerabilities that need to be addressed. Our design aims to provide a usable abstraction over the DHT-based overlay mesh network to aid developers in building privacy-friendly, censorship-resistant web applications.

3.2 Privacy-Enhancing Technologies

This section delves into a comprehensive analysis of privacy-enhancing technologies proposed and implemented in recent years, highlighting their significance and mechanisms.

Goldberg et al. [22] summarize the inventions of the early days of privacy-enhancing technologies, such as remailers, anonymizing web proxies, and anonymous digital cash. Remailers and anonymizing web proxies fundamentally rely on the concept of introducing anonymizing intermediaries between the sender and receiver, thereby obfuscating the origin of the communication. A widely deployed implementation of this approach is the Tor anonymity network by Dingledine et al. [23]. Every anonymity system requires ongoing research to develop defenses against novel attack vectors. For example Bahramali et al. [24] describe how deep neural networks can be trained to de-anonymize users of the Tor network. Danezis et al. [25] propose a provably secure message format called Sphinx. Building on top of Sphinx, Danezis et al. [26] propose HORNET, a high-speed mixnet. Kuhn et al. [27] discover a novel attack on the security proof of Sphinx and HORNET.

Encrypting centralized social media features, such as exchanging text and multimedia messages between two or more individuals, has become a well-established concept within digital communications. Schillinger et al. [28] describe a sophisticated web-based social network written in JavaScript that utilizes the crypto.subtle API[6]. This network enables encrypted messaging between individual users and groups, with all messages securely stored on a central server. They introduce the innovative concept of concealed addresses, which allows for the addressing of information while ensuring that the server does not learn contact relationships. This is achieved through the use of zero-knowledge proofs, which authorize the server to verify that a message can be accessed by a client without revealing the actual content. In their subsequent work, the authors tackle the issue of metadata leakage in encrypted social networks by employing onion-routing style "concealed channel" [29].

Dodson et al. [30] describe a social media application designed specifically for Android smartphones. In this system, central servers are used to host message queues of encrypted messages. The application assigns a public/private key pair to each device, enhancing

[6]`https://www.w3.org/TR/WebCryptoAPI/#Crypto-attribute-subtle` (last visited 2024-08-28)

security and privacy. The authors propose an intriguing method for key distribution by leveraging traditional social media platforms, such as Facebook. They provide APIs to manage common application styles, including turn-based games, where players take turns sequentially, which can operate on top of the social media application, thereby extending its functionality and increasing user engagement.

End-to-end encryption for specific applications has also seen innovative proposals. Hu et al. [31] focus on Google Docs, intercepting and encrypting messages used for communicating document updates via a web browser extension. This approach is particularly notable as it modifies the functionality of an existing application without requiring significant changes to the application itself, thereby maintaining user convenience while enhancing security.

A different approach to providing privacy to users was proposed by Wang et al. [32]. They propose a rule-based system for data usage and access, utilizing a client proxy and remote attestation, built on the existing TLS and DNS infrastructure. This system allows clients to define DNS domains permitted to process, aggregate, and store data, enforcing these rules through remote attestation. The authors suggest that organizations such as the EFF could define these rule sets, thus alleviating the need for end-users to invest significant effort and expertise in specifying rules.

Tor was initially designed for web-browsing, but its low-latency can even be used for phone calls as demonstrated by Bromberg et al. [33] who propose a Voice over IP (VoIP) system that employs Tor as its transport layer. By splitting the traffic over multiple Tor circuits, they achieve a higher quality of experience for users, addressing one of the common challenges associated with using Tor for VoIP.

Shafagh et al. [34] propose a decentralized data access control service designed for time-series data that can be selectively shared. Their approach leverages blockchain technology to store access control policies and uses a DHT-based storage network, ensuring both security and scalability.

Kogan et al. [35] present a system for private blocklist lookups. This system allows a client to determine if a particular string appears on a server-held blocklist without leaking the string to the server. They achieve this in sublinear time, significantly increasing server throughput by 6.7x, demonstrating the system's efficiency and practicality.

Khandelwal et al. [36] propose a privacy settings enforcement controller browser extension. This extension enables users to specify their privacy choices and automatically navigates the privacy menus of websites to enforce these choices. Additionally, they propose an automated cookie notice enforcer [37], which can disable all non-essential cookies through GDPR-mandated consent forms with a high success rate of 91%.

In privacy-enhancing distributed systems, synchronizing data across devices is a difficult problem. Khare et al. [38] propose extending the Representational State Transfer (REST) architecture for decentralized systems. Their approach allows agents within a distributed system to make decisions about data values based on estimations, using a centralized mutex to ensure total serialization of all updates to a resource. This approach enhances the consistency and reliability of decentralized systems.

One approach to synchronizing state across devices could be snapshotting state and transferring that snapshot to another device. Lo et al. [39] propose a generic method for migrating (i.e., storing and restoring) the current state of a website. They detail how function closures, event handlers, I/O events, and timers can be serialized, in addition to the trivially serializable state of a JavaScript-based web application, thereby preserving the full state of a web application across different sessions or environments.

Barradas et al. [40] describe PROTOZOA, a censorship circumvention system that utilizes covert channels over WebRTC video streams. PROTOZOA requires an out-of-band channel to exchange video-chatroom information, which is necessary to establish the WebRTC session and ensure that the communication remains hidden from censors.

Newman et al. [41] propose a high-bandwidth, metadata-private file broadcasting system. This system is designed to allow a small number of broadcasters to share a file with many subscribers, ensuring that the metadata remains private and secure.

The Tor project offers multiple censorship circumvention systems. One of them uses domain fronting, as described by Fifield et al. [42], and WebRTC data channels[7]. These mechanisms enhance the ability of users to bypass censorship and access information freely.

Smart Contracts on the Ethereum blockchain enable arbitrary code execution by anyone while presenting an eventually consistent world state [43]. There are multiple Ethereum clients, such as the Metamask web browser extension[8], which facilitate web applications' interaction with the Ethereum blockchain. The web browser extension is essential because the Ethereum network protocol cannot be directly implemented in a web browser.

Bowe et al. [44] extend this concept with ZEXE, which allows both code and data to be encrypted, thereby hiding them from third parties. A transaction in this system takes approximately 1 minute, in addition to the computation time required, highlighting its efficiency and security.

While Smart Contracts are a promising solution for managing distributed state replication, they face several unresolved issues that hinder their feasibility for general-purpose applications. These issues include the significant environmental impact associated with proof-of-work blockchains as described by Goodkind et al. [45] and the low transaction throughput as described by Bez et al. [46] and Samuel et al. [47], which currently limit their practical applications.

While there are many privacy-enhancing technologies available, the shift to smartphones, where software can only be obtained through a centrally controlled app store that

[7]`https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/wikis/Technical%20Overview`

[8]`https://metamask.io/` (last visited 2024-08-28)

sometimes delists applications upon government requests, can hinder the use of these technologies. This thesis aims to address this issue by providing a privacy-enhancing technology that can be accessed through a web browser on a smartphone, thereby enhancing the accessibility and availability of privacy-enhancing technologies.

3.3 Attacks on System Components

This section provides an in-depth discussion on various attacks targeting system components, particularly focusing on cryptography, DHT/mesh networks, and web browsers, which are relevant to our design.

3.3.1 Cryptography

In this subsection, we discuss related work concerning the cryptographic primitives employed in our design. Our selection of cryptographic primitives is based on the available cipher suites in modern web browsers (see subsection 4.3.1). We utilize AES in both GCM and CTR modes. For key exchange, we employ RSA and ECDH. For digital signatures, we use ECDSA.

Len et al. [48] demonstrate a partitioning oracle attack on AES-GCM by using a key multi-collision algorithm to find a collision in the GCM authentication tag. This attack highlights potential vulnerabilities in the implementation of AES-GCM and underscores the importance of rigorous cryptographic validation.

Bock et al. [49] investigated nonce reuse issues with the AES-GCM block cipher mode in TLS. They discovered that 184 HTTPS servers were repeating nonces and 70,000 servers were using random nonces, which can potentially lead to nonce reuse. This finding emphasizes the need for proper nonce management to ensure the security of encrypted communications.

Shakevsky et al. [50] uncovered an IV reuse attack on the TrustZone hardware in Samsung devices. The TrustZone is designed to protect cryptographic key material from being accessible to the main operating system, ensuring that keys are secure even if the main operating system is compromised. However, Samsung's implementation allowed an attacker to choose the IV of the AES-GCM algorithm within the TrustZone, enabling the extraction of wrapped keys and demonstrating a critical security flaw.

Jager et al. [51] describe a private key recovery attack on TLS-ECDH as described by Dierks et al. [52], which exploits faulty ECDH implementations that fail to validate whether a point is on a specific curve. This attack underscores the importance of proper validation in cryptographic implementations to prevent key recovery attacks.

Moghimi et al. [53] discovered an execution-time side-channel attack against TPM 2.0 devices deployed on commodity computers. They successfully recovered the 256-bit private keys used for ECDSA and ECSchnorr signatures, demonstrating the vulnerability of TPM devices to side-channel attacks and the need for improved security measures.

Alem et al. [54] describe a side-channel attack on OpenSSL's RSA implementation. They use analog signals produced by the processor during the execution of the RSA algorithm to extract key material, highlighting the risks associated with side-channel attacks and the importance of securing cryptographic implementations against such vulnerabilities.

Boneh et al. [55] provide a comprehensive overview of 20 years of attacks on the RSA cryptosystem, covering both elementary attacks on RSA's mathematical properties and implementation attacks.

Mus et al. [56] describe a key recovery attack for RSA and ECDSA using the Rowhammer attack as described by Kim et al. [57], a hardware bug that allows an attacker to flip bits in memory by repeatedly accessing nearby memory locations. This attack highlights the vulnerability of cryptographic systems to hardware-level exploits and the necessity of robust protection mechanisms.

Takahashi et al. [58] describe fault attacks against OpenSSL's implementation of elliptic curve cryptography. They demonstrate a full key recovery attack by injecting faults to create an EC key file with explicit curve parameters and a compressed base point, emphasizing the importance of securing cryptographic implementations against fault injection attacks.

Addressing metadata leakage from Signal's sealed sender protocol[9], Martiny et al. [59] propose ephemeral mailboxes, which enable two parties to exchange messages without disclosing their identity to the service provider. This approach relies on the assumption that the IP addresses of the communicating parties are hidden, highlighting a novel method for enhancing privacy in communication protocols.

Cherubin et al. [60] evaluate the viability of website fingerprinting attacks on the Tor network. Website fingerprinting utilizes machine learning techniques to identify websites based on the timing, volume, and direction of packets. They achieve above 95% accuracy when monitoring five websites, but the accuracy degrades to less than 80% when monitoring as few as 25 websites.

3.3.2 Web Browsers

Tim Berners-Lee et al. [61] describe the World Wide Web (W^3) model, a client-server architecture with three main components: (1) a common naming scheme for documents, (2) a common network access protocol, and (3) a common data format for hypertext. The W^3 client is also known as a web browser. Since its introduction in 1992 by Tim Berners-Lee et al. [61], the W^3, also known as the World Wide Web, has become widely adopted and has added capabilities for dynamic content with the introduction of JavaScript in 1995 [62]. This gradual evolution and wide adoption has lead to web browsers being a complex software system.

Squarcina et al. [63] found web application session integrity attacks by showing that the same-origin policy can be violated by a same-site (subdomains) cookie tossing attack.

[9]`https://signal.org/blog/sealed-sender/` (last visited 2024-08-28)

They describe how cookies, even HTTP-only cookies, can be evicted and overwritten by JavaScript. The "__Host-" prefix for cookies was intended to prevent this attack. However, an attacker can use nameless cookies and serialization collisions to circumvent these protections. Their research contributed to 12 CVEs, 27 vulnerability disclosures, and updates to the cookie standard, highlighting the ongoing challenges in maintaining web security.

Roth et al. [64] investigated inconsistent configurations of opt-in web browser security mechanisms sent by different web servers, such as HTTP Strict Transport Security (HSTS), a HTTP header signalign to the web browser that this origin should only be loaded with the TLS encrypted version of HTTP, HTTPs. They found that 321 out of 8,174 websites had inconsistent configurations either over time, between different web browsers, or from different source IP addresses. This inconsistency undermines the effectiveness of security mechanisms and emphasizes the need for standardized and consistent security practices.

Kim et al. [65] describe an attack vector to break the same-origin policy by leveraging web browser extensions that lack input validation and are poorly designed. They successfully extracted passwords and keys from password manager extensions, stole cryptocurrency, and performed universal cross-site scripting[10]. This study underscores the significant security risks posed by poorly designed browser extensions.

Lehmann et al. [66] describe binary exploitation of WebAssembly. WebAssembly allows compilation of memory-unsafe languages such as C and C++ to run in the web browser. Application vendors with large legacy codebases can compile their existing code to WebAssembly instead of rewriting it. However, WebAssembly does not offer the same protection mechanisms as native binaries, such as Address Space Layout Randomization (ASLR) as originally described by Forrest et al. [67], and stack canaries as described by Wagle et al. [68]. ASLR and stack canaries are techniques to make it more difficult to exploit memory corruption vulnerabilities. They found potential cross-site scripting attacks by exploiting WebAssembly programs, highlighting the need for enhanced security measures in WebAssembly implementations.

Eriksson et al. [69] analyze the security of web browser extensions and discover novel password-stealing, traffic-stealing, and inter-extension attacks. Furthermore, they identify 4,410 extensions that perform traffic-stealing attacks, emphasizing the pervasive nature of security vulnerabilities in web browser extensions and the need for improved vetting and security practices.

Calzavara et al. [70] describe inconsistencies in the click-jacking protection of web browsers. Modern web browsers support two different mechanisms for white-listing domains allowed to frame a website: X-Frame-Options and Content-Security-Policy. They found that 10% of the distinct framing control policies in the wild are inconsistent, highlighting the need for unified and consistent framing control policies to prevent click-jacking attacks.

[10]And you know it's a bad day if universal cross-site scripting is the third entry in such a list.

Gierlings et al. [71] describe abusing the site-isolation mechanism in web browsers to perform resource exhaustion attacks using fork bombs and UDP port exhaustion. Additionally, they demonstrate the feasibility of a DNS cache poisoning attack building upon their UDP port exhaustion attack with a success rate of 37%. This study underscores the importance of robust resource management and security mechanisms in web browsers.

Snyder et al. [72] describe a side-channel attack on modern web browsers that allows cross-origin communication between conspiring websites by exhausting limited resources managed by the web browser, such as WebSockets, web workers, and server-sent events. This attack highlights the potential for resource exhaustion to compromise web security and the need for improved resource management in web browsers.

Agarwal et al. [73] describe an exploit of the Spectre vulnerability in Chromium-based web browsers. This vulnerability breaks the site-isolation mechanism of the web browser, allowing an attacker to read the memory of other web pages as well as the LastPass password manager browser extension. This exploit demonstrates the pervasive nature of hardware vulnerabilities and the importance of mitigating such risks in web browsers.

Chinprutthiwong et al. [74] demonstrate the potential for XSS attacks to persist in a web browser using the service worker API. This study highlights the need for robust input validation and security practices to prevent persistent XSS attacks in modern web applications.

Mirheidari et al. [75] describe an improvement of the Web Cache Deception attack discovered by Mirheidari et al. [76]. They explain how misconfigured CDN caches can be exploited to retrieve private information from other users by crafting malicious URLs that trigger the caching of private information. This can be achieved, for example, by adding ".css" to the end of a URL. They propose a novel detection method and find 1,188 vulnerable websites, emphasizing the need for proper configuration and security practices in CDN caches.

The contribution of this master's thesis is to evaluate the feasibility of implementing privacy-friendly, censorship-resistant web applications in modern web browsers. Additionally, this thesis outlines an application architecture that does not rely on centralized data storage while maintaining the look and feel of traditional web applications. This is particularly relevant, as the traditional threat model for web applications assumes that the server is trusted and the client is untrusted, and web browsers have been designed around this threat model.

Our JavaScript framework, named Applink, aims to address the challenges associated with implementing the cryptographic layers of privacy-friendly, censorship-resistant web applications by providing a reference implementation that avoids pitfalls and vulnerabilities identified in the literature.

In this chapter, we have outlined related academically published work that is relevant to this thesis. In the next chapter, we will discuss the fundamentals of the technologies used in this thesis.

CHAPTER 4

Fundamentals

This chapter provides a comprehensive overview of the essential concepts and technologies necessary for implementing a distributed application framework. The discussion covers critical definitions, security principles, web browser capabilities, and relevant APIs for cryptography, data storage, and communication.

4.1 Definitions

To ensure clarity and consistency throughout this thesis, we define several key terms:

(i) Peer: An application running in a web browser that participates in the network.

(ii) Contact: A peer included in the contact list of another peer.

(iii) Trusted Peer: A peer that is trusted by a contact. This property is transitive, meaning if Peer A trusts Peer B and Peer B trusts Peer C, then Peer A trusts Peer C. This forms a web of trust model. The transitivity of trust may be limited to a certain depth to prevent extending trust to all peers in the network.

4.2 The CIA Triad

The CIA triad, as described by Rhodes-Ousley [77], is a foundational model used to evaluate the security of a system. The triad comprises three core principles:

1. Confidentiality: Ensures that information is accessible only to those authorized to have access. The system must prevent unauthorized access to sensitive information.

2. Integrity: Ensures that information remains accurate and unaltered during storage, transmission, and processing. The system must prevent unauthorized modifications to information.

3. Availability: Ensures that information and resources are accessible to authorized users whenever needed. The system must prevent unauthorized denial of access to information.

While availability is generally understood to mean that a system or service can be connected to and used, it is equally important to consider social availability. Social availability refers to the awareness and acceptance of particular software by normal users and the social implications of its usage. For example, if the use of certain software is criminally punishable, even if not actually enforced, it affects the software's social availability.

4.3 Browser Capabilities

This section provides an overview of the relevant web browser APIs associated with cryptography, persistent data storage, and communication that are currently available in modern web browsers. These capabilities are essential for implementing secure and efficient distributed applications.

4.3.1 Cryptography

Web browsers implement the SubtleCrypto interface, which provides a set of methods for performing various cryptographic operations. These functionalities include symmetric and asymmetric encryption, creating and verifying digital signatures, and key generation.

The W3C WebCrypto specification recommends that the following algorithms should be implemented[1]:

```
HMAC using SHA-1
HMAC using SHA-256
RSASSA-PKCS1-v1_5 using SHA-1
RSA-PSS using SHA-256 and MGF1 with SHA-256.
RSA-OAEP using SHA-256 and MGF1 with SHA-256.
ECDSA using P-256 curve and SHA-256
AES-CBC
```

However, in practice, web browsers support a broader range of algorithms[2]:

```
RSASSA-PKCS1-v1_5, RSA-PSS, ECDSA, HMAC,
RSA-OAEP, AES-CTR, AES-CBC, AES-GCM,
SHA-1, SHA-256, SHA-384, SHA-512,
ECDH, HKDF, PBKDF2,
AES-KW
```

[1] https://www.w3.org/TR/WebCryptoAPI/#algorithm-recommendations-implementers (last visited 2024-08-28)

[2] https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto (last visited 2024-08-28)

The storage of key material is not specified by the WebCrypto API, and it may be extractable by other, non-privileged processes on the operating system outside the web browser[3] if stored in the persistent storage of the web browser. One of our research questions is to investigate the possibility for web applications to use the WebAuthn API to encrypt key material before storing it in persistent storage.

WebAuthn

The W^3C WebAuthn specification provides a signature-based authentication scheme where an authenticator signs a challenge with a private key. According to the specification, the authenticator can be implemented in software, on a secure coprocessor, or on a physical token accessible via USB, Bluetooth, or NFC[4].

A Pseudo Random Function (PRF) is a cryptographic function that is indistinguishable from a random oracle for a polynomial-time adversary. A random oracle is a function that maps every value in the possible input set to a value in the possible output set, returning a random value for each input but always the same value for the same input. It is impossible for an adversary to predict the output value from the input value without access to the oracle.

A web application can use the proposed PRF Extension to WebAuthn to encrypt data in the web browser by generating a random seed and storing it in plain text. When the application is opened, the PRF provided by WebAuthn derives an encryption key from the seed, which is then used to encrypt data stored in persistent storage[5].

4.3.2 Persistent Storage

Web browsers offer multiple methods for storing data, with the primary differences being storage duration and the ability to structure data.

SessionStorage and LocalStorage

SessionStorage enables developers to store key-value pairs in storage managed by the web browser until the page is closed. Both keys and values must be of the string type[6]. SessionStorage can be used to store data across multiple tabs but is not suitable for long-term storage, as it gets cleared when the page is closed.

LocalStorage is similar to SessionStorage but with one key difference: LocalStorage data has no expiration time, meaning it persists even after the web browser is closed and reopened, while SessionStorage data is cleared when the page session ends[7].

[3]`https://www.w3.org/TR/WebCryptoAPI/#security-developers` (last visited 2024-08-28)
[4]`https://www.w3.org/TR/webauthn-2/` (last visited 2024-08-28)
[5]`https://w3c.github.io/webauthn/#prf-extension` (last visited 2024-08-28)
[6]`https://html.spec.whatwg.org/multipage/webstorage.html` (last visited 2024-08-28)
[7]`https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage` (last visited 2024-08-28)

IndexedDB

IndexedDB is a built-in database that allows for storing structured data within the web browser. It operates as a key-value store capable of storing JavaScript objects. IndexedDB is asynchronous and uses transactions to ensure data consistency. It also supports storing binary data. However, by default, the data is not encrypted and can be accessed by other, non-privileged processes on the operating system outside the web browser[8].

Cache

The W^3C created the ServiceWorker specification to allow developers to create web applications that can continue to function while a device is unable to communicate with the applications web server after the application has cached the necessary resources. Caches are part of the W^3C ServiceWorker specification and allow developers to store key-value pairs where the key is a URL string of the resource and the value is a response object. This enables the storage of both text and binary data[9].

File System API

The File System API allows web applications to read and write files to the user's local file system. It offers access to the file system of the device by letting the user select files or directories and read or write files. Additionally, it offers the Origin private file system, which is a sandboxed file system that is unique to the origin of the web application[10]. Safari and Firefox only implement the Origin private file system, while Chrome and Edge implement both the Origin private file system and the user's file system[11].

4.3.3 Data Exchange

This section provides an overview of the relevant web browser APIs associated with data exchange that are currently available in modern web browsers.

Origin

The origin of a web page is defined as the combination of the protocol, domain, and port of the URL. The Same Origin Policy restricts web applications to interact with resources from different origins. This policy is enforced by web browsers to prevent malicious websites from accessing sensitive data from other websites[12].

[8]`https://www.w3.org/TR/IndexedDB/` (last visited 2024-08-28)
[9]`https://www.w3.org/TR/service-workers/#cache-objects` (last visited 2024-08-28)
[10]`https://fs.spec.whatwg.org/` (last visited 2024-08-28)
[11]`https://developer.mozilla.org/en-US/docs/Web/API/File_System_API` (last visited 2024-08-28)
[12]`https://www.w3.org/Security/wiki/Same_Origin_Policy` (last visited 2024-08-28)

Fetch Requests

Fetch requests allow JavaScript to make HTTP(s) requests, enabling the retrieval of resources from web servers. This mechanism is restricted by the Same Origin Policy[13].

WebRTC

Web Real-Time Communication (WebRTC) was originally designed to enables web applications to capture and optionally stream audio and video and arbitrary media between browsers. WebRTC provides an encrypted peer-to-peer connection between web browsers. To establish a connection, the two communicating web browsers need to exchange two messages, an offer and an answer. The specification omits how these messages are exchanged. WebRTC offers the option to proxy traffic through turn servers [78] to allow web browsers to bypass strict NAT rules[14]

Push API

The Push API[15] enables sending push notifications to a web browser. Upon receiving a push notification, the web browser starts the Service Worker[16] of the corresponding website, even if the website is not currently open, and dispatches the “push” event. This mechanism allows for arbitrary code execution within the service worker on demand without user interaction.

The Push API[17] allows the transmission of arbitrary data, which is encrypted using Elliptic Curve Diffie-Hellman on the P-256 curve and a symmetric secret combined with the ECDH key using the HMAC-based key derivation function [79].

Because the HTTP endpoints for sending push notifications block CORS requests (until September 2022, Mozilla Firefox accepted CORS requests to its Push Server[18]), websites cannot directly send push notifications and must go through a web server that strips the CORS header.

To send push notifications, the sender must have a push registration from the recipient. The push registration includes the recipient’s p256dh public key, the recipient’s auth secret, and the recipient’s endpoint URL. The sender encrypts the payload using the recipient’s public key and then sends the encrypted payload to the recipient’s endpoint URL.

[13] `https://fetch.spec.whatwg.org/#fetch-method` (last visited 2024-08-28)
[14] `https://www.w3.org/TR/webrtc/` (last visited 2024-08-28)
[15] `https://www.w3.org/TR/push-api/` (last visited 2024-08-28)
[16] `https://www.w3.org/TR/service-workers/` (last visited 2024-08-28)
[17] `https://www.w3.org/TR/push-api/#dom-pushsubscription` (last visited 2024-08-28)
[18] `https://github.com/w3c/push-api/issues/303` (last visited 2024-08-28)

WebSockets

WebSockets provide a full-duplex communication channel between a client and a server over a single, long-lived connection [80]. This allows for real-time data exchange and reduces latency compared to traditional HTTP requests. WebSockets can be used for signaling in WebRTC as well as for other real-time communication needs in a distributed application framework.

Various server-side technologies support WebSockets, including Node.js, Python, Java, Ruby, Go, and .NET[19].

4.3.4 WebAssembly

WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine[20]. It is designed as a portable target for the compilation of high-level languages like C, C++, and Rust, enabling deployment on the web for both client and server applications[21]. WebAssembly can be used to run computationally intensive tasks at near-native speed in web browsers[22].

4.4 Sender-Recipient Unlinkability

A desirable property of a system is to prevent adversaries from learning which people are communicating with each other. When applications communicate over a network it is possible for an adversary to observe how packets are flowing through the network for example by writing down every packet passing through a network switch. The naive solution to prevent an adversary from learning the recipient of a network packet is to sent the packet to every possible recipient at once. The sender of such a packet is still known of course.

Kuhn et al. [81] introduce the concept of sender-recipient unlinkability (M_{SR}), representing an adversary's inability to determine the sender or recipient of a message within a network. A Chaumian mixnet, as described by Chaum [82], can achieve M_{SR} by routing messages through a series of intermediary nodes, also known as relays, each of which only knows the previous and next node in the chain.

A widely adopted implementation of such a system is Tor – the onion router by Dingledine et al. [23], which has been in development since 2002. Tor routes packets through a "circuit", a series of intermediary nodes. Tor uses three nodes by default for a circuit. Tor calls the first intermediary node a client connects to the "entry" node, and the last node (typically the third node) the "exit" node. Because the entry node knows the IP

[19] `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API` (last visited 2024-08-28)

[20] `https://webassembly.org/` (last visited 2024-08-28)

[21] `https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts` (last visited 2024-08-28)

[22] `https://www.w3.org/TR/2022/WD-wasm-web-api-2-20220419/` (last visited 2024-08-28)

address of the client, the entry node is chosen from a preselected pool of nodes called "guard" nodes. The guard nodes are chosen by the client and are kept for a long time to avoid malicious nodes being selected as the entry node. Web servers can use the Tor network to hide their IP address by running a "hidden service" on the Tor network.

Tran et al. [83] discuss compromising DHT-based onion routing by skewing relay selection towards malicious peers. This threat can be countered by using a directory of trustworthy onion routers from the peer set, necessitating a central authority for directory maintenance.

Schuchard et al. [84] propose a DHT-based relay selection algorithm that attempts to prevent skewed relay selection by having peers vouch for the authenticity of routing tables by signing them.

Erdin et al. [85] have identified 16 classes of attacks on anonymity networks such as onion routing, seven of which are network-level attacks and the remaining nine are application-level attacks. According to Erdin et al., application-level attacks occur through the use of applications that do not take privacy into account during implementation. As this work is concerned with the implementation of a privacy-focused application framework, we will focus on the network-level attacks.

1. Intersection Attacks: An attacker can observe the set of active users at a given time interval and identify communicating users by observing the intersection of the sets.

2. Flow Multiplication Attacks: An attacker controlling the entry and exit nodes of a circuit can introduce, modify, or drop packets (or cells) and observe this change at the other end of the circuit. This allows the attacker to deanonymize the source and destination of a circuit.

3. Timing Attacks (also known as flow correlation attacks): The timing attack observes the traffic volume and timing of tunneled connections to correlate the sender and recipient. Bahramali et al. [24] describe how deep neural networks can be trained to automate this attack.

4. Fingerprinting Attacks: The fingerprinting attack described by Erdin et al. [85] is so similar to the timing attack that we will not distinguish between the two.

5. Congestion Attacks: An attacker can observe the traffic at an exit node and flood other nodes with traffic to determine if a particular node is part of a circuit.

6. Resource Attacks: An adversary can set up a large number of relay nodes to increase the probability of being selected as a node in a circuit. This allows the adversary to observe more traffic and potentially deanonymize users.

7. Denial of Service Attacks: An ISP can block access to publicly known Tor relays or attempt to overwhelm the relays with traffic (DDoS) to prevent users from accessing the network. The Tor Project has developed a countermeasure to blocking relays by using bridges, which are not publicly known relays that can run various protocols.

Evers et al. [86] have identified attacks on the Tor implementation of onion routing. These attacks mostly overlap with the findings of Erdin et al. [85]. Evers et al. [86] describe two additional classes of attacks on onion routing:

1. Supportive Attacks: These attacks do not directly aim to deanonymize users or disrupt the overlay network but rather aim to support other attacks. 1.1 The guard selection of Tor can be manipulated by an adversary with access to the client's path to the entry node (e.g., ISP). The adversary can block all but one guard node a user connects to, forcing a new round of guard selection in the hope that an adversary-controlled relay will be selected as a guard node. 1.2 The sybil attack involves an adversary creating a large number of nodes in the network to increase the probability of being selected as a node in a circuit.

2. Revealing Hidden Services: An adversary can attempt to become the first node on the path of a hidden service to gain a direct connection to the hidden service and reveal its IP address. Additionally, an adversary can compare the clock skew of a hidden service and a normal web server running on the same machine to reveal the location of the hidden service.

In this chapter, we have discussed the CIA triad, web browser capabilities, and the concept of sender-recipient unlinkability. We have also reviewed the attacks on onion routing networks and the countermeasures that have been proposed. In the next chapter, we will discuss the requirements for Applink.

CHAPTER 5

Requirements Analysis of the Privacy-First Web Application Framework

As this work aims to investigate the possibility of implementing applications using a privacy-preserving, partially replicated, decentralized, peer-to-peer JavaScript framework, it is essential to define realistic requirements for the sample applications to identify any obstacles to implementation with currently available web browser APIs.

This chapter outlines the requirements for Applink based on the requirements of the sample applications. First, we define some general requirements for all sample applications as well as specific requirements for each application in section 5.1. Then, we outline the requirements for Applink in section 5.2. Finally, we describe the user stories for the sample applications in section 5.3.

5.1 Sample Web Applications

This section specifies the requirements for three sample applications.

In addition to the specific requirements of the sample applications, the following requirements shall be met by all sample applications:

For a decentralized framework to be adopted, it needs to be comparable to or better in terms of user experience than a centralized web application.

The application shall perform within the constraints established by Miller [87] on a smartphone:

1. The application shall respond to user input within 100ms.

2. The application shall draw a frame within 16.6ms (60fps).

3. The application shall load within 5000ms.

Furthermore, the application shall always present a consistent (but possibly stale) state across the entire user interface:

1. Data shall be consistent within a page[1].

2. Data shall be consistent across page navigation.

3. Data shall be consistent across multiple tabs of the same application within a web browser.

The application shall inform the user in case an action fails, provide a reason why, and give a recommendation on how to proceed.

These general requirements ensure the user experience is comparable to that of a centralized web application.

5.1.1 Note Application

Users should be able to use the note application to take notes for themselves, share them with friends and family, and synchronize them between their devices. This application is a good fit for Applink because there are few interactions with other users and therefore few synchronization events that can lead to inconsistent data across different users.

The following requirements are based on the use-case diagram of the note application (see Figure 5.1).

1. The application shall allow a user to create, update, and delete a note.

2. The application shall allow a user to view their notes.

3. The application shall allow a user to share their note with other users ($0..*$).

4. The application shall allow a shared note to be edited by all users it has been shared with.

5. The application shall allow a user to synchronize their notes with other devices.

6. The application shall not allow users to view, modify, or delete the notes of other users without explicit permission.

[1]This is relevant when multiple elements on a page are based on common data. For example, a chart of money spent this month shall update when a user enters a new transaction.

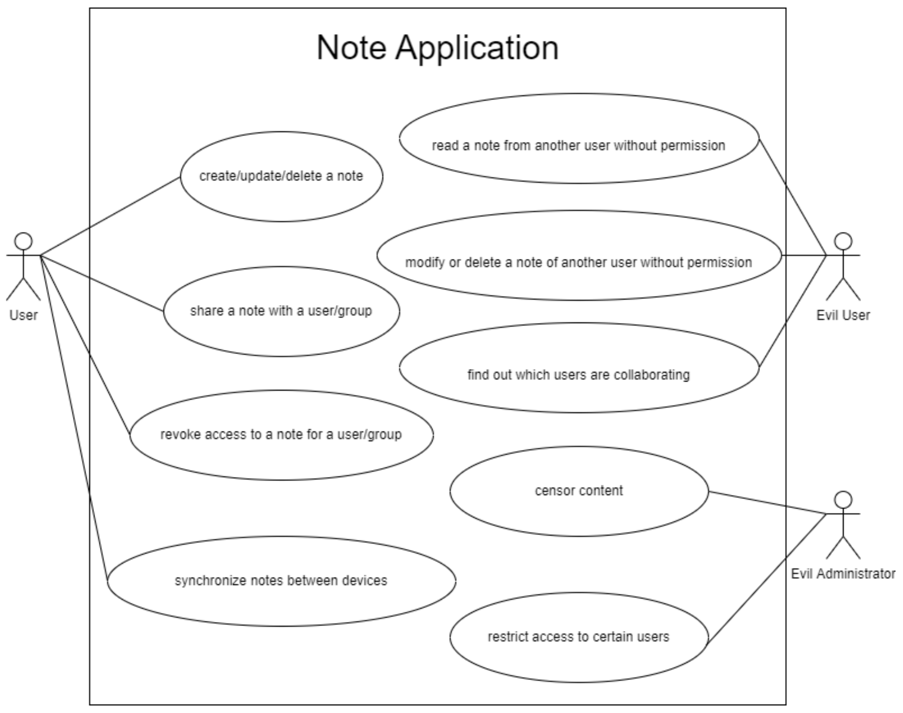

Note Application
User
create/update/delete a note
share a note with a user/group
revoke access to a note for a user/group
synchronize notes between devices
read a note from another user without permission
modify or delete a note of another user without permission
find out which users are collaborating
Evil User
censor content
restrict access to certain users
Evil Administrator

Figure 5.1: Use-Case Diagram Note Application

7. The application shall not reveal the set of users with access to a note to users without access to the note.
8. The application shall not allow anyone to prevent anyone else from using the application.

5.1.2 Beer Credit System

In a student bar, students can deposit money ahead of time and buy drinks using their deposit. To keep track of these deposits, we implement the “Beer Credit System (BCS)”. Deposits must be approved by an administrator. The BCS is a poor fit for our Applink Javascript Framework as most of the data needs to be validated by a centralized group of administrators. This application is included in the set of prototype applications to tests edge cases of Applink.

The following requirements are based on the use-case diagram of the beer credit system (see Figure 5.2).

1. The application shall allow a user to read their account balance.
2. The application shall allow a user to request a deposit.
3. The application shall allow a user to buy a drink.

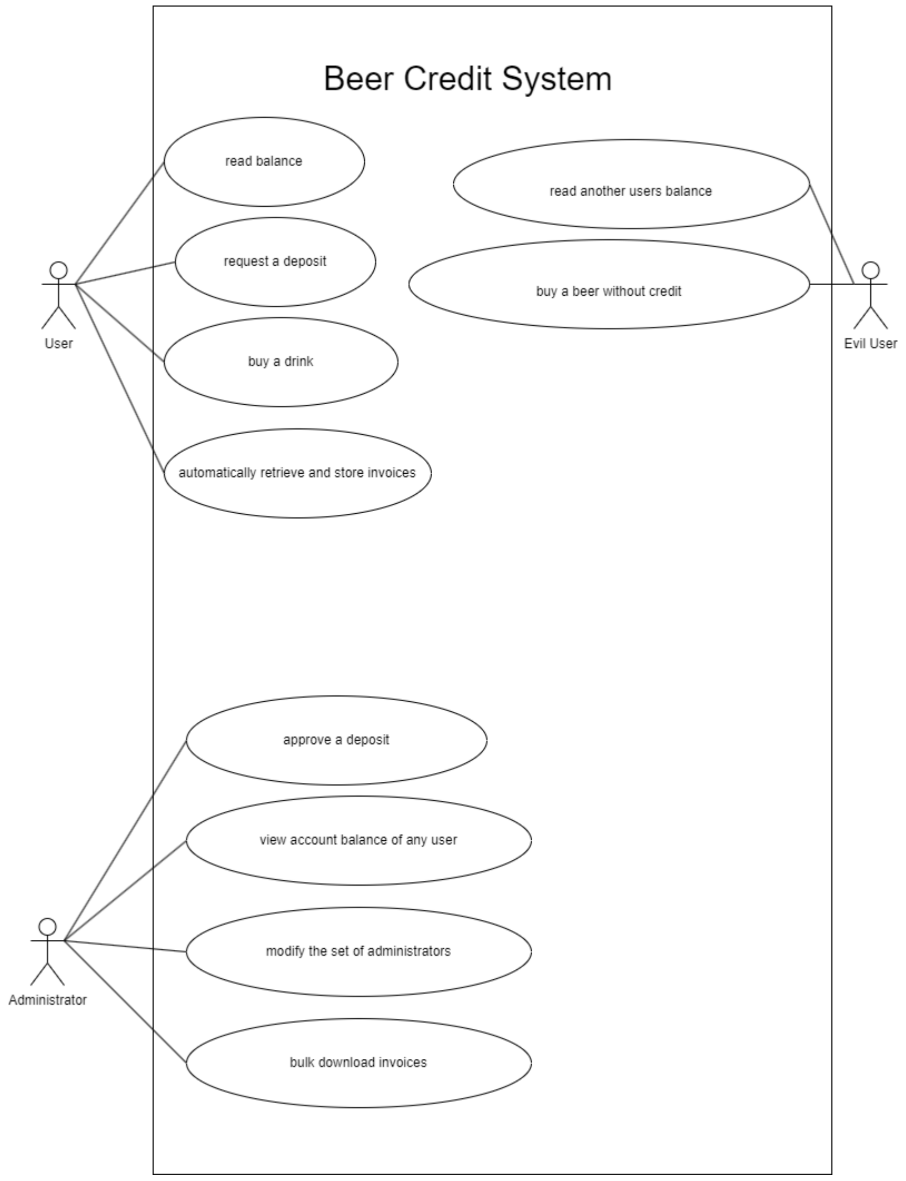

Beer Credit System
read balance
request a deposit
buy a drink
automatically retrieve and store invoices
read another users balance
buy a beer without credit
User
Evil User
approve a deposit
view account balance of any user
modify the set of administrators
bulk download invoices
Administrator

Figure 5.2: Use-Case Diagram Beer Credit System

4. The application shall allow a user to retrieve and store invoices in an invoice application of choice.

5. The application shall allow administrative users to approve a deposit.

6. The application shall allow administrative users to view the account balances of any user.

7. The application shall allow administrative users to modify the set of administrators.

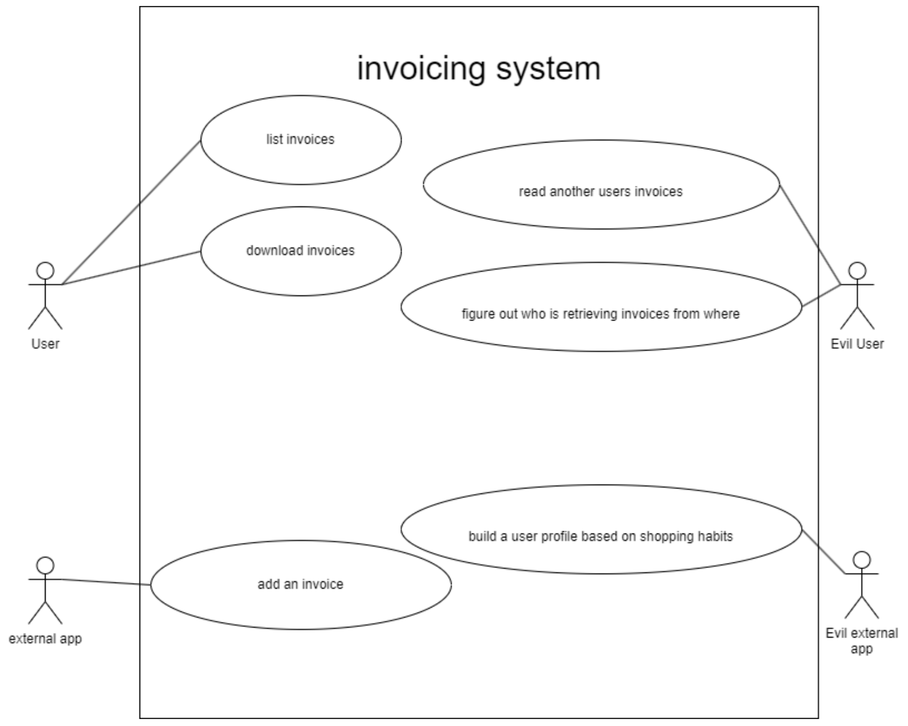
8. The application shall allow administrative users to bulk download the already existing invoices.

9. The application shall not allow a user to read another user's balance.

10. The application shall not allow a user to buy a drink with insufficient credit.

5.1.3 Invoicing System

The beer credit system (see subsection 5.1.2) shall send invoices in a machine-readable format to the invoicing system. The invoicing system can then display detailed information, such as the sum of individual goods purchased per time period (e.g., week). The invoicing system demonstrates how different applications can exchange data without sacrificing the goals of Applink.


invoicing system
list invoices
read another users invoices
download invoices
User
figure out who is retrieving invoices from where
Evil User
build a user profile based on shopping habits
add an invoice
external app
Evil external app
Figure 5.3: Use-Case Diagram Invoicing System

The following requirements are based on the use-case diagram of the invoicing system (see Figure 5.3).

1. The application shall allow a user to view a list of their invoices.

2. The application shall allow a user to view an invoice.

3. The application shall allow an external application to store an invoice on behalf of the user.
4. The application shall prevent a user from reading invoices issued to other users.
5. The application shall offer anonymity for the sender and recipient of invoices for other users of the system and external observers.

5.2 Requirements

This section outlines the requirements for the Applink JavaScript framework based on the requirements of the sample applications.

5.2.1 Functional Requirements

The functional requirements are:

1. Applink shall provide an appropriate interface for persisting and retrieving data.
2. Applink shall provide a signal when data is modified. This signal should always be sent when the data in the persistence layer is modified, even if it happens by another tab or other context with access to the same origin.
3. Applink shall provide the ability to replicate its state to other web browsers.
4. Applink shall provide the ability to remotely call functions of other users.
5. Applink shall provide a mechanism for determining the caller of a remote procedure call. This indication is meant for authorization checks and shall therefore be guaranteed to be authentic. The duty of authorization lies with the application code.
6. Applink shall provide a mechanism for pairing a new browser with the role of the other browser being administrator, another user, or the same user.

5.2.2 Threat Model for Privacy-First Web Applications

This section outlines our threat model for privacy-first web applications. Our threat model considers three roles: censors, malicious users, and malicious administrators.

We assume censors to have read access to all centralized infrastructure such as web servers. Additionally, censors control some nodes in the peer-to-peer network and some fraction of routers on the network. Censors can perform active and passive attacks against the network. Censors can order ISPs to block certain websites using IP and DNS blocking. When publishers use a CDN to host static content, censors can only use DNS blocking as IP blocking would result in over-blocking [88]. Furthermore, we assume censors will not shut down the push-notification infrastructure as this would cause too much disruption.

Malicious administrators have access to all data on the web server. They can turn off the server entirely, either voluntarily or involuntarily[2]. Administrators might want to discriminate against certain information or users and might want to build micro-targeting profiles of users.

Malicious users do not have access to centralized data. They can use active and passive attacks against other users. They can send arbitrary data to arbitrary users and observe all traffic routed through nodes controlled by them. The attack goals of malicious users are to access information of other users without permission or to identify the contacts of specific users[3].

We assume that for every user there exists a set of trusted contacts. These trusted contacts are assumed to not perform active attacks and are assumed to be less likely to be malicious compared to other users participating in the overlay network[4]. The contacts trusted by trusted contacts can also be presumed to be more trustworthy compared to unknown peers, allowing the construction of a web of trust.

Metadata and Side-Channels

We define user data to include not just data that is entered by users into the application or collected from other sensors by the application, but also metadata that is not actively collected by the application code. When an application sends a packet, the content of the packet is not the only data an attacker can collect. The attacker can also collect the time the packet was sent, the size of the packet, the address of the sender and recipient, etc. When the timing of certain events unintentionally discloses information, this is called a side-channel. We consider metadata and side-channels to be accessible by passive and active attackers and therefore needs to be addressed by Applink.

5.2.3 Cross-Application Data Exchange

Applications often need to exchange data between them. For example, an application might import a friend list from a different application or share a link via a messaging service.

Traditional web applications can use CORS requests[5] to communicate with other applications. CORS requests require a central server to function and therefore once again reintroduce the privacy drawbacks this framework aims to avoid. Therefore, a different mechanism for data transmission between web applications on different origins is required.

[2]Censors might force a project to stop. e.g., the NSA appears to have forced TrueCrypt to stop development [89]

[3]e.g., an ex-boyfriend wants to figure out who his ex-girlfriend is dating now

[4]As creating malicious peers that are trusted by a user is considerably more difficult but not impossible

[5]https://fetch.spec.whatwg.org/#cors-request

5.2.4 Performance Requirements

Based on the constraint of applications to draw a frame within 16ms to render 60 frames per second, Applink needs to take up less than 16ms minus the time it takes the web browser to draw a frame to allow the web browser to render the page at 60 fps. If Applink exceeds 16ms of uninterrupted JavaScript execution, the web browser cannot draw a frame in time. Therefore, Applink shall not cause JavaScript execution of more than 10ms.

Users experience two different scenarios when it comes to the synchronization of data. Either the user has two devices with the web application open at the same time, or the user makes changes on one device and then later looks at the same data on a different device.

If a user has the web application open on two devices at the same time, the expectation is that the changes propagate in real-time. This work considers a delay of two seconds acceptable for this scenario. Two seconds are chosen because it provides an acceptable delay for collaborative tasks.

5.2.5 Developer Usability Requirements

A framework should improve the developer experience when building decentralized web applications. Any framework should only include abstractions that are easier to use than building equivalent functionality from scratch.

Developers need a way of keeping the user interface in a consistent state. Multiple components of the user interface might depend on the same database value. Therefore, Applink needs to notify all components that accessed a value when that value is modified or deleted.

Transactions are essential to any application. Their success or failure might trigger complex correction algorithms or just display a message to a user. The webpage might be closed by the user during a transaction. The promise[6] callback system is not suitable for this kind of transaction logic because if the JavaScript virtual machine is closed, the context of the promise (e.g., the remaining promise chain[7], the Closure[8]) is lost. Therefore, Applink needs to provide a way to handle transactions that are not lost when the JavaScript virtual machine is closed.

Applink shall provide a way to call functions running in a different web browser and respond to the result. This remote procedure call syntax shall be easily understandable because developers need to reason about their program to avoid bugs and vulnerabilities.

[6]`https://tc39.es/ecma262/multipage/control-abstraction-objects.html#sec-promise-objects`

[7]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining` (last visited 2024-08-28)

[8]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures` (last visited 2024-08-28)

5.2.6 Economic Requirements

Applications built with Applink shall not burden the organizations or individuals running them with high infrastructure costs. Hosting of a web application can never be zero because the static assets of a web application (html, JavaScript, css) need to be served to the users. In order for the costs to stay sustainable this work defines the following upper limits:

The costs of hosting the server logic of an Applink application shall not be more than the costs of serving the static content (html, JavaScript, css) of the application by more than 200%. E.g. if it costs 1€ to host the static content of an application, it shall not cost more than 3€ to host the entire application including all server side logic.

5.3 User Stories

This section describes the user stories for the sample applications.

5.3.1 User Stories: Note Application

Here, we outline the user stories derived from the requirements of the note application.

1. As a user, I want to create a note so that I can write down my thoughts.
2. As a user, I want to edit a note so that I can correct mistakes or add additional information.
3. As a user, I want to delete a note so that I can remove notes I no longer need.
4. As a user, I want to share a note with another user so that I can collaborate with them.
5. As a user, I want to see notes shared with me so that I can read them.
6. As a user, I want to see notes shared with me updated in real-time so that I can see changes made by other users.

5.3.2 User Stories: Beer Credit System

Here, we outline the user stories derived from the requirements of the beer credit system.

1. As a user, I want to see my account balance so that I can see how much money I have left.
2. As a user, I want to request a deposit so that I can buy drinks.
3. As a user, I want to buy a drink so that I can quench my thirst.

4. As a user, I want to retrieve and store invoices in an invoice application of choice so that I can keep track of my expenses.
5. As an administrator, I want to approve a deposit so that I can prevent users from buying drinks with insufficient credit.
6. As an administrator, I want to view the account balances of any user so that I can see how much money users have left.
7. As an administrator, I want to modify the set of administrators so that I can add or remove administrators.
8. As an administrator, I want to bulk download the already existing invoices so that I can archive them.

5.3.3 User Stories: Invoicing System

Here, we outline the user stories derived from the requirements of the invoicing system.

1. As a user, I want to view a list of my invoices so that I can see how much money I have spent.
2. As a user, I want to view an invoice so that I can see the details of a specific invoice.
3. As a user, I want external applications to be able to store an invoice on my behalf so that I can import invoices from other applications.

In this chapter, we defined the requirements for the sample applications and the Applink JavaScript framework. In the next chapter we will discuss the system architecture of Applink.

CHAPTER 6

System Architecture of the Privacy-First Web Application Framework

As explained in the threat model in subsection 5.2.2, we assume that censors and malicious administrators have access to data stored on centralized infrastructure. One solution to prevent centralized authorities from being able to access the personal data of all users of a system is to encrypt all data using a key accessible only to the intended recipient. However, this approach still allows the centralized authority to control the permitted communication links between users and the ability to ban users from the service. Additionally, centralized data storage burdens a central entity with infrastructure costs, which may become unsustainable when a system is used by a large number of users. Consequently, we opted for a peer-to-peer architecture for our framework, assuming that lower operating costs and ease of deployment would benefit both administrators and users.

6.1 Overview

Applink offers two main features: an abstraction for data storage based on IndexedDB described in section 6.2 and an RPC interface described in section 6.3 for peer-to-peer communication.

The RPC interface uses WebRTC with Web Push as a signaling channel described in section 6.4.

The WebRTC connections allow message routing using a DHT based on the k-bucket[1] library. Peers can perform further WebRTC handshakes over the established WebRTC

[1]`https://github.com/tristanls/k-bucket` (last visited 2024-08-28)

connections to minimize reliance on the centralized Web Push infrastructure. To choose the correct parameters for the DHT, we implemented a simulation described in section 6.5.

To mitigate the risk of impersonation and session correlation, we use time-based addresses described in section 6.6 for addressing peers in the DHT.

The cryptographic layer uses ECDH-AES-GCM for link encryption within the DHT layer and RSA-OAEP-SHA256 for message encryption for intended recipients. An additional application encryption layer, accessible only to direct contacts using permanent keys, ensures further privacy. We describe the encryption and communication process in section 6.9.

We also implemented encryption of data stored in persistent storage using the WebAuthn PRF extension described in section 6.10.

The details of our approach to anonymity are described in section 6.11.

An overview of the communication architecture is shown in Figure 6.1. The application code interacts with the dht-transport-manager (Dht TM), which translates from contact addresses to time-based addresses described in section 6.6 and is responsible for bootstrapping the DHT by establishing some initial WebRTC connections via the Web Push API.

The Dht TM then injects the bootstrapped connections into the Dht class, which is responsible for maintaining the DHT and routing messages. The Dht class establishes further WebRTC connections to peers discovered in the DHT.

The WebRTC connections are managed by the WebRTC class, which is responsible for containing the implementation details of establishing and maintaining the WebRTC connections.

6.2 Data Storage Abstraction

Because web browsers have a key-value database, called IndexedDB, built in, and the only other options for storing data in a web browser are localStorage which is synchronous and has a small storage limit, and the cache api which does not guarantee that the data will be preserved, we chose to use IndexedDB as the data storage for our framework.

We wanted to provide a storage abstraction that feels similar to an orm for the developer because we believe that developers are used to working with ORMs and that this would make it easier for developers to write code that runs in the web browser. We also needed this abstraction to store the data the framework needs to function.

This data storage abstraction automatically migrates the database when the web application is loaded, creating all stores that are declared by the framework and developers and dropping any stores that are no longer needed.

We define a BaseDto class that specifies the structure of the data that can be stored in IndexedDB. The BaseDto defines an id field that is used as the key for the corresponding

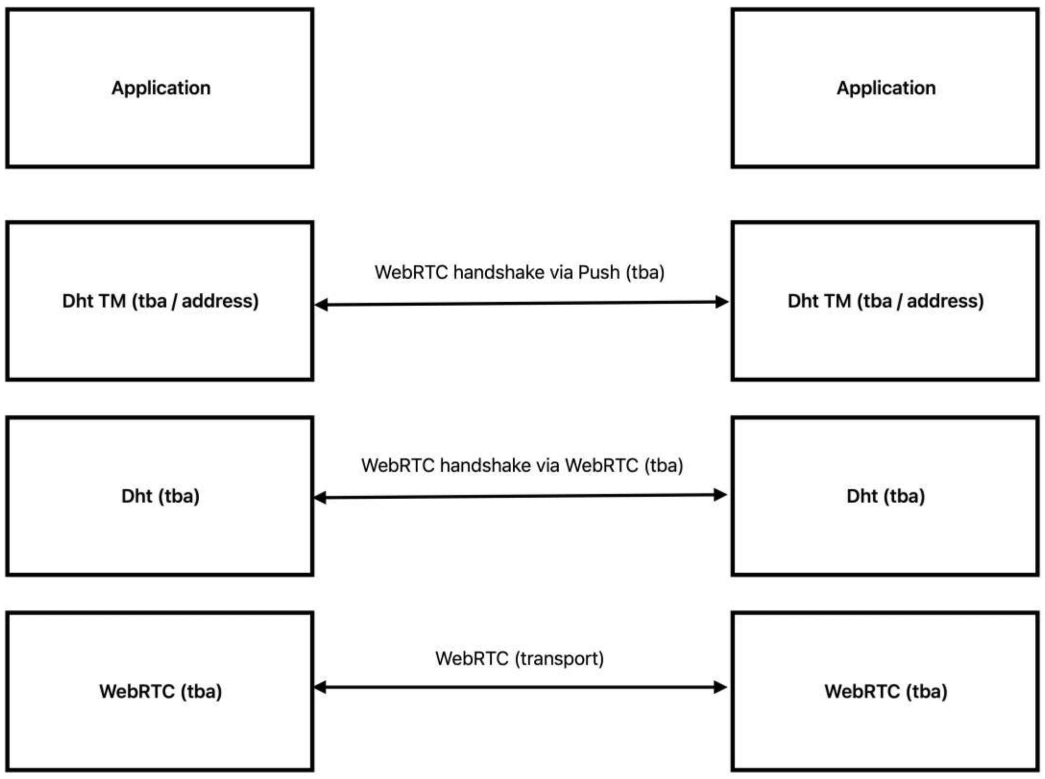


Figure 6.1: Overview of communication architecture.

IndexedDB object store as well as a static dbName field that defines the name of the object store. The BaseDto definition is shown in Listing 6.1.

Listing 6.1: BaseDTO definition

```
export interface BaseDto {
    id?: string;
}

export interface DtoClass {
    new(...args: any[]): BaseDto;
    get dbName(): string;
}
```

The data storage is initialized using the DbFactory class, which automatically creates the object stores for the Dtos that are registered with it. The TransactionManagers init function seen in Listing 6.2 is responsible for checking which stores already exist in the web browsers IndexedDB (line 6), then deletes the stores that are no longer needed (lines 8-12), and creates the stores that are not present in IndexedDB (lines 13-17). The DbFactory class creates a transactionManager object that creates all the object stores for the Dtos that are registered with it and removes object stores that are no longer needed. IndexedDb only calls the upgrade function when the version of the database is increased. Because the framework did not want to rely on the application developer to increase a version number when the schema changes because this is easy to forget, the DbFactory class uses the current time as the version number. This ensures that the upgrade function is called every time the application is loaded.

The DbFactory class needs to be a singleton because it opens the IndexedDB database, which can only be done once per origin at a time. If this is attempted a second time, the promise to open the IndexedDB will hang indefinitely.

Listing 6.2: Creating IndexedDB object when the framework is initialized when the application is loaded

```
init(dbName: string): Promise<void> {
    return this.initPromise = (async () => {
        const tmThis = this;
        this.db = await openDB(dbName, new Date().getTime(), {
            upgrade(db) {
                const existingStores = Array.from(db.objectStoreNames);
                // Synchronize the declared object stores with IndexedDB
                for (const oldName of existingStores) {
                    if (!tmThis.stores.includes(oldName)) {
                        db.deleteObjectStore(oldName);
                    }
                }
                for (const storeName of tmThis.stores) {
                    if (!existingStores.includes(storeName)) {
                        db.createObjectStore(storeName, {
                            keyPath: 'id',
                            autoIncrement: false
                        }).createIndex('id', 'id', { unique: true });
                    }
                }
            },
        });
    })();
}
```

The database initialization logic is abstracted away, and to use it, we can simply call the getDao method of the DbFactory class with the DtoClass that we want to use and then call the CRUD methods of the ObjectStore that is returned, as shown in Listing 6.3.

Listing 6.3: PushTransport.ts

```
this._pushTransportStore =
    await this.dbFactory
        .getDao<ObjectStore<PushTransportDto>>(PushTransportDto);
await this._pushTransportStore!.add(new PushTransportDto(userId, address));
```

This abstraction also supports transactions. This is a non-trivial feature to implement because IndexedDB closes a transaction when the microtask queue is empty. Therefore, we need to collect all the operations that are part of a transaction and execute them in a single microtask. If code has to wait for information from a previous operation in the transaction, the transactionManager provides a wait function that returns a promise that resolves when the previous operations have finished. This allows the transactionManager to resolve all dependencies of a transaction, such as cryptographic operations, network requests, etc., before starting a transaction and then executing the entire transaction within a single microtask.

An example of a transaction is shown in Listing 6.4. In this example, we add a contact and reference the contact from the contact address store. To get the ID of the contact, we need to wait for the contact to be added to the contact store before adding the contact address. To do this, we use the wait function provided by the transactionManager.

Listing 6.4: Running transactions

```
await this.dbFactory.transactionManager
  .runTransaction(async (dbTransaction, wait) => {
    //await other promises here first

    this.contactDao!.add(contactDto, dbTransaction)
    (async()=> {
        await wait();
        this._contactAddressDao!
          .add(contactDto.id, ownAddressDto, dbTransaction);
    })();
});
```

The implementation of the runTransaction method, shown in Listing 6.5, first awaits the callback function, resolving all dependencies of the transaction and collecting the operations that can be performed synchronously. Then it iterates over the entries of the transaction, executing the onExecute defined by the CRUD methods of the ObjectStore class. If an error occurs, the transaction is aborted and the error is thrown.

Listing 6.5: Implementation of runTransaction

```
async runTransaction(
    callback: (dbtransaction: Transaction, wait: CallableFunction) => void)
    : Promise {
    await this.lock();
    try {
        if (!this.initPromise) {
            throw new Error('TransactionManager not initialized');
        }
        await this.initPromise;
        if (!this.db) {
            return;
        }
        const transaction = new Transaction();
        const wait = async () => {
            while (transaction.hasNextEntry()) {
                await transaction.getNextEntry().onExecute();
                }
            };
            await callback(transaction, wait);
            const tx = this.db.transaction(
                Array.from(transaction.entityNames), 'readwrite');
            try {
                while (transaction.hasNextEntry()) {
                    await transaction.getNextEntry().onExecute();
                }
                await tx.done;
            } catch (e) {
                console.error(e);
                await tx.abort();
                throw e;
            }

            const postTransactionStateList =
              (await Promise.all(transaction.completeEntries
                .filter(entry => entry.onGetPostTransactionState)
                .map(entry => entry.onGetPostTransactionState?.())
                .filter(entry => entry !== undefined)))
            for (const { listener, type } of this.dataChangeListeners) {
                for (const entry of postTransactionStateList) {
                    if (entry.id?.startsWith(type.dbName + ':')) {
                        listener(entry);
                    }
                }
            }
        }
        finally {
            this.unlock();
        }
    }
```

This abstraction allows for promise-based atomic transactions that can be used in the RPC interface to ensure that the data is in a consistent state across peers.

6.3 RPC Abstraction

The abstraction chosen for our framework is a Remote Procedure Call (RPC) interface that facilitates calling functions on other peers identified by a unique ID. To communicate with a peer, one must know the peer's address, which consists of the unique ID and additional metadata enabling the establishment of a secure connection. This address can be serialized and shared in a manner deemed appropriate by the developer. For instance, the information could be exchanged out of band (e.g., using a link) or through mutual peers.

Once a peer is known, developers can invoke functions on that peer, akin to interacting with smart contracts on the Ethereum blockchain. The code is written from a peer's perspective and may permit calls from other peers.

To inform developers that a function is accessible remotely, functions registered with the `RPCInterface` abstraction must start with "_r_". This should allow developers to easily identify which functions are accessible remotely and account for this when writing and reviewing code.

In Listing 6.6, we illustrate an implementation of such a calling structure. The function `importDocumentShare` initiates the calling chain by invoking `documentShared` on the owning peer of the document. It is important to note that the call to `documentShared` specifies an expected callback to `documentSharedAck`. This mechanism enables developers to define which callbacks they expect to be triggered by an RPC call. If the callback to `documentSharedAck` is not executed within the specified timeout, the call to `documentShared` will result in an error. In such cases, the call to `documentShared` is retried until successful. This allows exceptions to be handled by the invoking function even if the error occurs in a callback, as the stack trace is preserved by the `RPCInterface`.

To illustrate this, imagine an interaction between Peers A and B. `importDocumentShare` on Peer A calls function `_r_documentShared` on Peer B, which in turn calls `_r_documentSharedAck` on Peer A. If function `_r_documentSharedAck` fails, the stack trace contains `_r_documentSharedAck` and `importDocumentShare` and the invoking stack trace of the call to `importDocumentShare`. This allows developers to better understand exceptions and what caused them.

Every invocation of a function includes the userId of the invoker. Authenticity and integrity are provided by the application encryption layer as described in section 6.9. Functions can only be invoked by contacts known to a peer.

`RPCInterface` allows type checking with the TypeScript compiler, making it easier to write correct code. The TypeScript compiler can validate the function name and the parameters in the call to a function and the function that is registered to be called.

However, a malicious user might still pass incorrect types to the function. Developers should therefore validate the type of the parameters of the function on top of other input validation before using them. An example of this is shown in Listing 6.7. The function "f" is defined to take a parameter "a" and return the length of "a". However, multiple types implement the "length" property. In the first call, a string is used, in the second call, an array, and in the third call, an object with a "length" property. Note that the third call returns an array instead of an integer.

The TypeScript compiler does not add checks to the function body to ensure that a parameter is of a certain type. This can lead to vulnerabilities and should be a consideration in development and code review processes.

Listing 6.6: NoteStore.ts

```
interface NoteShareParameterMap {
    'documentShared': { noteId: string, challenge: string };
    'documentSharedAck': { noteId: string };
    'documentUpdated':
      { noteId: string, updateVector: string, origin: string };
}

export class NoteShareController {
//...
constructor() {
    this._initPromise = this._init();
}

async _init() {
    //...
    this.rpci =
        new RPCInterface<NoteShareParameterMap>(this.signalDht, 'notes');
    this.rpci.register('documentShared', this._r_documentShared.bind(this));
    this.rpci.register('documentSharedAck',
        this.r_documentSharedAck.bind(this));
    this.rpci.register('documentUpdated',
        this._r_documentUpdated.bind(this));
}


async importDocumentShare
    (from: string, docId: string, challenge: string, isSameUser = false) {
   //...
    await this.noteShareDao!
        .add(new NoteShareDto(docId, from, challenge, isSameUser));
    //...
    /* retry sharing until successful */
    while (true) {
        try {
            /* call documentShared on the owner of the document
             * we expect documentSharedAck to be called within 10 seconds,
             otherwise we retry
            */
            await this.rpci!
            .call('documentShared', from, { noteId: docId, challenge },
            { functionName: 'documentSharedAck', timeout: 10000 });
            /* the documentSharedAck was called, we can break */
            break;
        } catch (e) {
            console.error(e);
        }
    }
}


async _r_documentShared
    (
        from: string,
        { noteId, challenge }: NoteShareParameterMap['documentShared']
    ) {
    await this._initPromise;
    const noteShares = await this.noteShareDao!.getAll();
    const existingNoteShare = noteShares
        .find(noteShare => noteShare.noteId === noteId
            && noteShare.userId === from);

    /* if the note is already shared with the user,
     * we still tell the user in case they didn't get the ack last time
     */
    if (existingNoteShare) {
        await this.rpci?.call('documentSharedAck', from, { noteId });
        return;
    }
    const note = noteShares
        .find(noteShare => noteShare.noteId === noteId
            && noteShare.challenge === challenge
            && noteShare.userId === null);
    if (!note) {
        throw new Error(`got invalid documentShared`);
        return;
    }
    note.userId = from;
    note.challenge = null;
    await this.noteShareDao!.update(note);
    //...
    /* this rpc call does not include an expected callback
    * as it is the end of this exchange
    */
    await this.rpci!.call('documentSharedAck', from, { noteId });
    /* send the current state of the document to the new user */
    await this.rpci!.call('documentUpdated', from,
        { noteId, updateVector: noteVector, origin: null! });
}

async r_documentSharedAck
    (from: string, { noteId }: NoteShareParameterMap['documentSharedAck']) {
    await this._initPromise;
    const note = (await this.noteShareDao!.getAll())
        .find(noteShare => noteShare.noteId === noteId
            && noteShare.userId === from);
    if (!note) {
        console.error(`got invalid documentSharedAck`);
        return;
    }
    note.challenge = null;
    await this.noteShareDao!.update(note);
    if (!note.isSameUser) {
        this.documentListUpdateListener?.(UpdateType.ADD, noteId);
    }
    else {
            this.updateSameUserAddresses();
        }
}}
```

Listing 6.7: JS Function Parameter vulnerability

```
> var f = (a)=>a.length
undefined
> f('asdf')
4
> f([1,2,3,4])
4
> f({'length':[1,2,3,4]})
[ 1, 2, 3, 4 ]
```

6.4 Peer-to-Peer Bootstrapping

As outlined in section 4.3.3, WebRTC necessitates a two-way handshake, facilitated through an auxiliary channel, to establish a connection.

Applications using a web server to establish peer connections face potential censorship by ISPs, either through server shutdowns or DNS-record modifications to block server access.

To mitigate reliance on a single central server, applications can utilize the Web Push API (push), deemed too critical to block, for bootstrapping. Once a node establishes a WebRTC connection with another node, it can tunnel additional handshakes through the existing connections.

As seen in the sequence diagram Figure 6.2, a peer A starts the bootstrapping process by sending a push notification containing a handshake request to one of its known contacts, B. When a user launches the application for the first time, they do not have a contact B to perform the bootstrapping with because they do not know any push registrations. Our solution to this is to require an out of bands “invite” link that contains the push registration of the inviter. This way, the new user can establish a connection to the inviter and then establish connections to the inviter’s contacts.

If the recipient is currently online, it responds with a push notification containing a handshake that includes WebRTC signaling information.

The initiating peer, A, then sends another push notification with the handshake response, including its WebRTC signaling information.

After these initial three push notifications, WebRTC can establish a connection between the two peers. To finish the initial connection establishment, peer B sends a connection established call to peer A via WebRTC. This call lets peer A know that the connection has been established and that peer B is ready to receive messages.

After the initial WebRTC connection has been established, the DHT class automatically discovers other peers using the “FindNode” call. Upon learning of new peers from the “FoudNode” call, the DHT class will attempt to establish WebRTC connections to the new peers until the connection limit is reached.

Because the handshake calls via WebRTC are sent at a time when peer A does not know its address neighborhood, regular DHT routing of messages will not work, which is why the HandshakeWith call includes a respondVia field. In our example, peer C would then wrap its response in a layered packet akin to onion routing, with the outer packet addressed to peer B and the inner packet addressed to peer A.

Given WebRTC’s inherent need for a two-way handshake via centralized infrastructure, it’s impossible to conceal application usage. If, as in the case of our framework, web push is used as a signaling channel for WebRTC, peers need to know the push registration of some participants in the network. This can either be well-known peers, which would

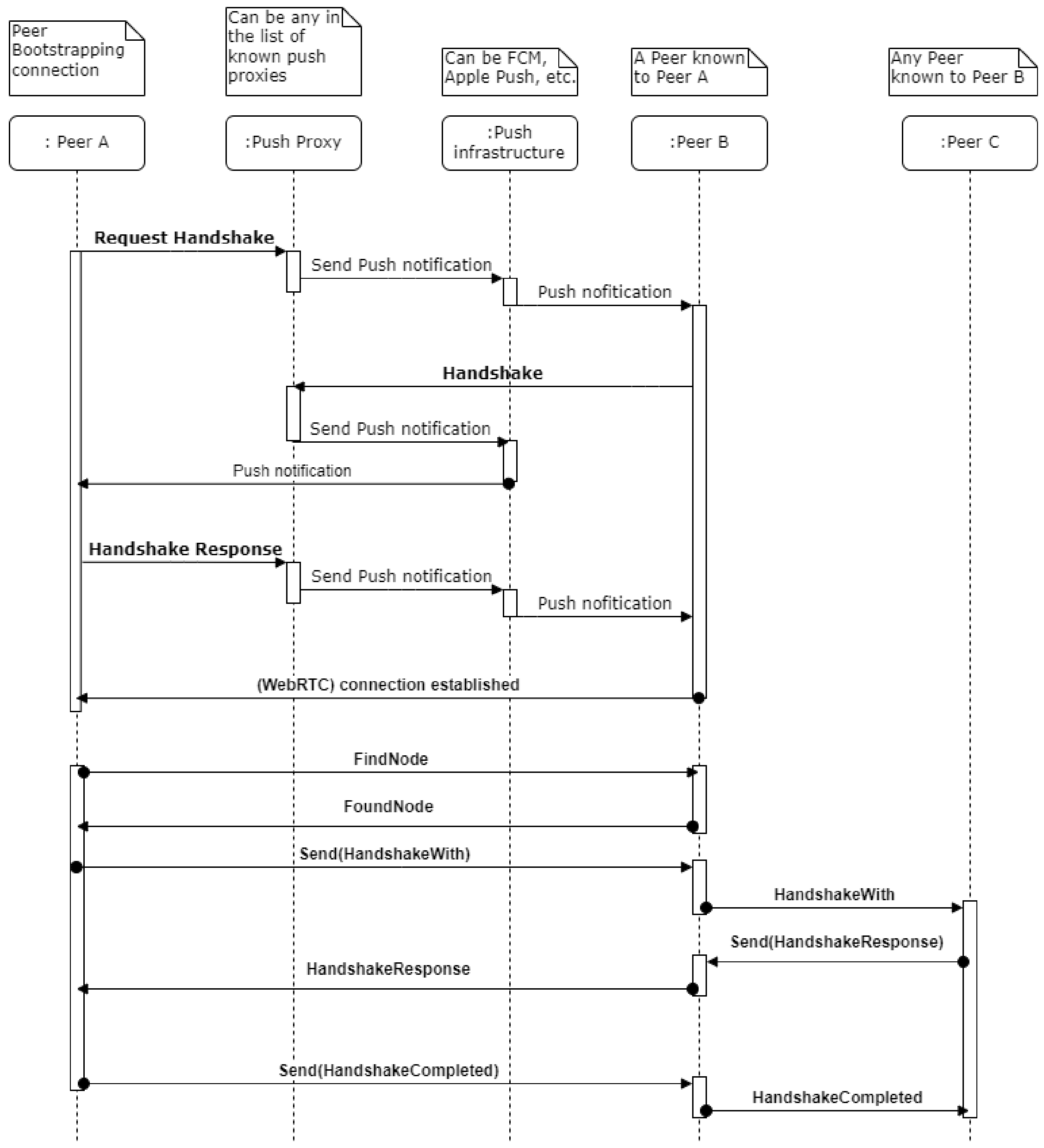


Figure 6.2: Connection Establishment Sequence Diagram

introduce reliability on those peers being online, or the push registrations of their contacts, which reveals a subset of every peer's contacts to the push server when they establish a connection using push.

Push provides the benefit that it is assumed to not be taken down. Even if some servers that we use to proxy traffic to the push infrastructure are taken down, it is enough that every peer can reach at least one server to establish a connection. Furthermore, if clients

A and B want to connect to each other, they do not need to be able to connect to the same proxy server. This means that even if we have only two proxy servers and client A can only connect to the first and client B can only connect to the second, they can still establish a connection.

In September 2022, Mozilla Firefox ceased accepting CORS requests to its Push Server[2], complicating Push API-based connections. We propose circumventing this limitation through several HTTPS proxies that relay messages to the Push API, operated by volunteers. Applications can incorporate functionalities to automatically distribute these proxies among trusted contacts.

After a connection to an initial peer is established, a peer can use the WebRTC DHT to find and connect to other peers. This reveals to everyone who can query the DHT a list of peers that are online. This list can be used to track the online status of a peer. Additionally, if a malicious actor wants to impersonate a peer and analyze who is trying to communicate with them, there is nothing stopping them. We propose mitigating these problems using time-based addresses as described in section 6.6.

6.5 Simulation and the Importance of Correct Bootstrap Parameters

We wrote a simulation to determine the parameters for the bootstrapping process as well as the DHT. We wrote it in JavaScript and used the force-graph[3] library to visualize the resulting network.

The simulation mimics the behavior of the DHT network with a few simplifications: Instead of using time-based addresses (see section 6.6), we use random addresses because it does not impact the DHT behavior, as we can just hold a reference to all contacts of a peer instead of that peer needing to come up with a pseudo-random identifier that only their contacts can find. We assumed that every peer knows five contacts on average and we simulate a population of 300 peers. We assume that in every step of the simulation, every peer has a 1 percent chance to go online or offline.

When a peer goes online, it will try to establish a connection analogous to DhtTransportManager: It first connects to N known contacts, then connects to up to 20 more peers. The first N connections are simulated as if they were push notifications. They select a random contact from the address book, and the connection is only successfully established if the contact is online in that tick. The additional peers have to be connected to peers that are already connected to the peer. This simulates the DHT behavior where a peer can only connect to peers that are in their address neighborhood.

To find the number N, we ran the simulation with different values of N and observed the resulting graph. In Figure 6.3, we observe that with N=1, the network tends to form

[2]`https://github.com/w3c/push-api/issues/303` (last visited 2024-08-28)
[3]`https://github.com/vasturiano/force-graph?tab=readme-ov-file` (last visited 2024-08-28)

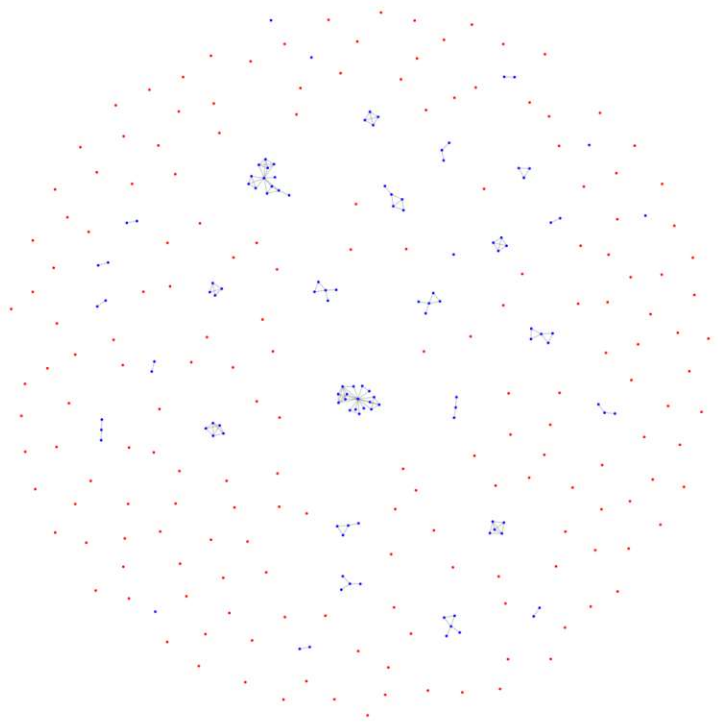


Figure 6.3: Simulation of the DHT network with N=1, blue=online, red=offline.

islands of connectivity. This appears to happen because once a peer establishes the first connection, it can only discover peers that are in that initial connected cluster.

When we configure peers to connect to two known contacts before connecting to additional peers, the network is almost fully connected as seen in Figure 6.4. This appears to be because every peer has a chance of bridging the gap between two clusters of peers.

Comparing the two simulations, we can see that the choice of N has a significant impact on the connectivity of the network. We can conclude that the choice of N should be at least 2 to ensure that the network is well connected. Because the initial connections to contacts require the push notification service, we should keep N as low as possible. We set N to 2 in the prototype implementation.

The simulation can be paused and resumed to observe snapshots of the network at any given moment in time.

We further extended the simulation to estimate the number of successful message deliveries in the network. We describe these extensions in subsection 7.3.3.

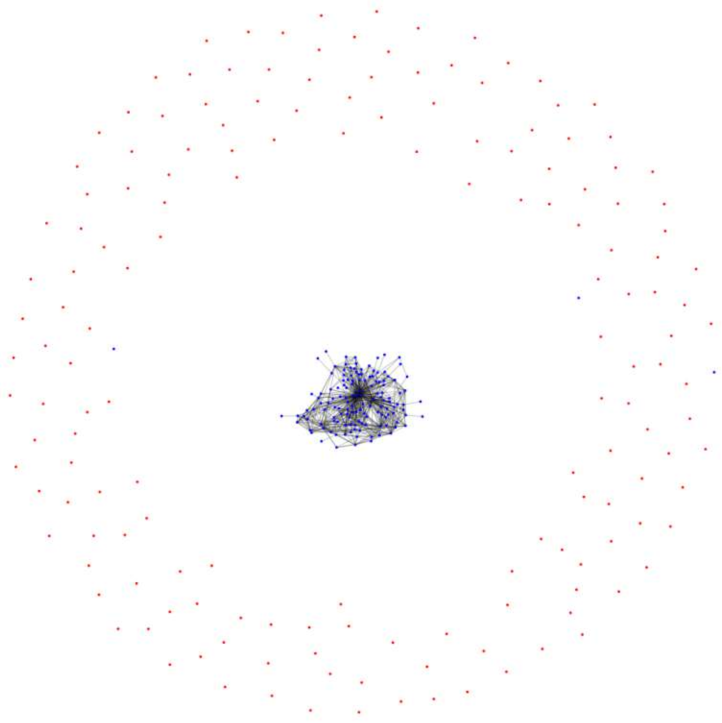


Figure 6.4: Simulation of the DHT network with N=2, blue=online, red=offline.

Another extension we made to the simulation was to add malicious peers to simulate the impact of honest-but-curious attackers. We describe this extension in subsection 7.2.5.

6.6 Time-Based Addresses

When users' addresses do not change over time, it is easy for an attacker who controls some peers in the network and observes messages sent through the network to correlate the sessions of users.

Such an attacker can observe which peers are listed in the DHT and therefore online at the same time and count packets exchanged between them. Because users' addresses do not change, such an attacker has plenty of time to observe communication patterns and build a complete social graph of the network.

To frustrate the attacker described above attempting to build social graphs of the overlay network, we introduce time-based addresses. A time-based address, inspired by the Time-Based One-Time Password (TOTP) algorithm [90], is generated by hashing the concatenation of the user ID and a timestamp. This process yields a transient, pseudo-random address to use in the DHT.

Definition:

$$A_t = \mathrm{H}(A_u + floor(\frac{t}{I})) \tag{6.1}$$

Where:

A_t is the time-based address at time t

A_u is the user ID

t is the current time

I is the interval at which the address A_t changes (in our implementation, 24 hours)

H is a secure hash function where it is impossible for a polynomial time-bounded adversary to find the preimage of a hash

Time-based addresses decouple the user ID from the DHT address. Their time dependence and the fact that they rotate all at the same time across the network prevent a passive attacker from correlating sessions of users not in their contact list based on observing the DHT entries. An active attacker could still connect to a peer directly and obtain a WebRTC handshake containing the IP address of that peer.

In case of a leak of the A_u, all A_t are compromised. To mitigate this, a user should be given the option to migrate to a new A_u with a new DHT signing key and inform their contacts of the new A_u and signing key. This mechanism is not implemented in the prototype but would essentially involve creating a new A_u and corresponding signing key and broadcasting the change to all online contacts. The contacts would then update their address book with the new A_u and key. This broadcasting to contacts would have to continue until all contacts acknowledge the change because not all contacts might be online to receive the message. This message must be signed with the old A_u's key; otherwise, the contacts would not accept the message.

The time-based address allows peers that have the contact information of a peer to find them in the DHT while not allowing adversaries without access to the contact information to do the same.

When receiving a message, every peer needs to determine which peer it came from to verify the message signature. To do this, we generate a rainbow table containing all time-based addresses of all contacts upon loading the application.

6.7 Unlinking Time-Based Addresses From Static Identifiers

Joining the overlay network requires exchanging web push registrations, which are linked to a specific web browser on a specific device and generally do not change over time and are linkable to a specific user account by the push server. For example Google Chrome uses Firebase Cloud Messaging (FCM) which is is owned by Google. Therefore if a user uses Google Chrome, Google can link the push registration to the user's Google account.

We consider the scenario, where a user wants to share their push-notification registration with untrusted peers to give those untrusted peers the ability to come online even if none of their contacts are online. When such an untrusted peer then connects to the user described above, the untrusted peer then learns the relationship between the users time-based address and the push registration. If this happens multiple times, the untrusted peer can link the users time-based addresses over time.

Even in a scenario, where only trusted users are given the push registration of the user described above, because the bootstrapping of new peers always starts at one of their contacts, a passive network observer can observe the contact relationship between the two time-based addresses.

To mitigate this, we propose a double onion circuit bootstrapping process.

Consider peers A and B. peer A wants to join the network and peer B is already connected to the network. First, peers A and B exchange their push notifications and establish a WebRTC tunnel. Peer B then establishes an onion circuit to peer C and tunnels all packets of peer A through peer C. peer A then establishes an onion circuit to peer D and tunnels all his packets through peer D. This way, both peer A and B can be certain that their initial connection can not be linked to their time-based addresses.

Only at this point does peer A reveal his time-based address to the network and establishes a new WebRTC connection to a fresh peer E. After establishing the connection to peer E, peer A drops his connection to peer B and the onion circuits to peers C and D are closed. Peer A is now connected to the network and neither peer B learned the time-based address of peer A nor did peer A learn the time-based address of peer B. Furthermore, as far as a passive network observer is concerned, peer A just appeared in the network at a random peer who is unrelated to peer A.

This process is illustrated in Figure 6.5.

6.8 Censorship Resistance vs Anonymity

Web applications using WebRTC as a transport layer and Time-Based Addresses as described in section 6.6 still need to consider that directly connected peers can observe each other's IP addresses. To avoid this, the application can tunnel their WebRTC traffic through a turn server and filtering the WebRTC signaling information to exclude IP addresses that are not the turn server. WebRTC offers web applications the option to tunnel the WebRTC communication channel through a turn server to bypass restrictive NAT configurations. A web application can specify a list of turn servers it wants to use. We can abuse this feature to hide the IP addresses of the peers from each other.

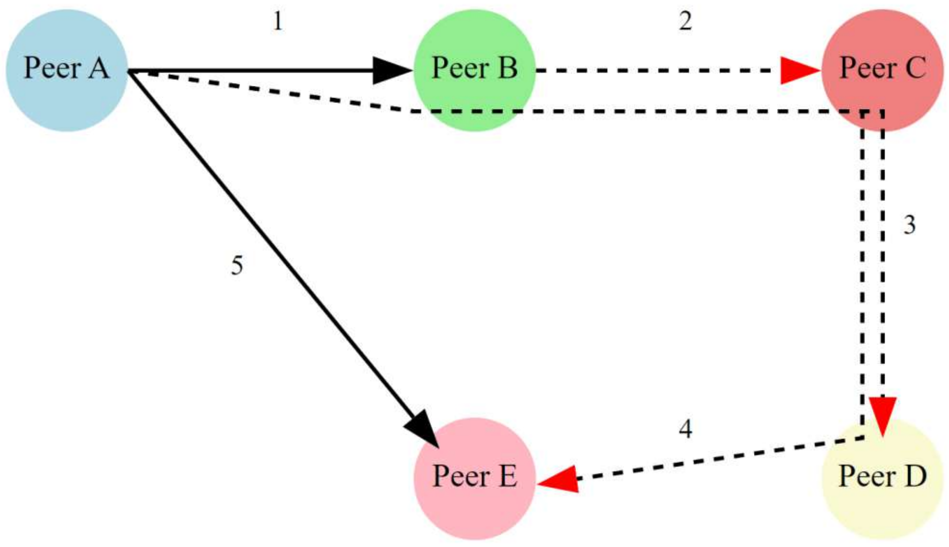


Figure 6.5: Double Onion Circuit, dottet-line: onion circuit, solid line: WebRTC connection

However, when using a turn server, the turn server can potentially be shut down by an adversary, preventing communication or forcing the application to reveal the IP address of its network connection.

6.9 Encryption and Communication

As peers periodically alter their addresses, corresponding updates to their DHT encryption keys are essential. Each user's DHT record includes their time-based address, public key, and a signature of this public key, authenticated with a permanent RSA-PSS-SHA256 keypair. The public key of this RSA-PSS-SHA256 pair, shared only with direct contacts, validates the signature of the peer's session encryption keys. Employing ECDH for this purpose would be inadvisable due to its potential for public key recovery, linking time-based addresses[4].

An overview of the encryption and communication process is shown in Figure 6.6.

For link encryption within the DHT layer, we employ ECDH-AES-GCM with a dynamic key authenticated with the RSA-PSS-SHA256 key. Message encryption for intended

[4]`https://www.secg.org/sec1-v2.pdf` (last visited 2024-08-28)

recipients utilizes RSA-OAEP-SHA256 with the dynamic key also authenticated with the RSA-PSS-SHA256 key.

Beyond the DHT layer, an additional application encryption layer, accessible only to direct contacts using permanent keys, ensures further privacy. This layer adopts either ECDH-AES-GCM or the Double Ratchet Algorithm as implemented by the Libsignal protocol[5] or ECDH-AES-GCM.

As the Double Ratchet Algorithm requires a handshake for each communication partner, it makes key distribution more complex. Think of a note application, where a note can be shared with a group of users. By using the ECDH-AES-GCM encryption layer, all public keys of the group members can be stored in the metadata of the note. This way, updates to the note can be shared with all group members without requiring a key-exchange protocol to have taken place between all members of the group. The Double Ratchet Algorithm, on the other hand, would require an “introduction” of the new group member to all other group members. Alternatively, every group member could share a list of prekey bundles in the metadata, but this would still risk two new group members picking the same prekey bundle, causing a conflict.

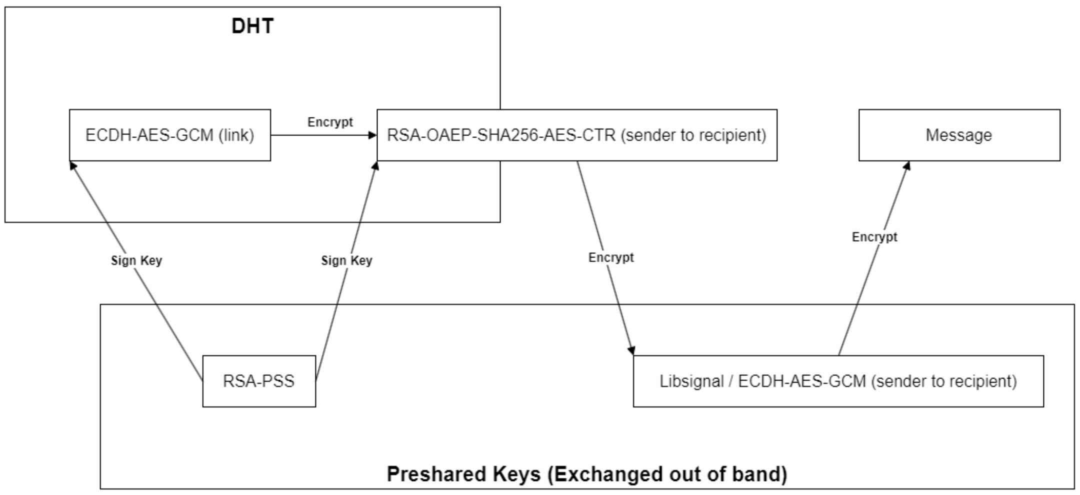

DHT
ECDH-AES-GCM (link)
Encrypt
RSA-OAEP-SHA256-AES-CTR (sender to recipient)
Message
Sign Key
Sign Key
Encrypt
Encrypt
RSA-PSS
Libsignal / ECDH-AES-GCM (sender to recipient)
Preshared Keys (Exchanged out of band)

Figure 6.6: Encryption Diagram

6.10 Encryption of Persistent Data

To prevent other non-privileged applications running on the operating system from accessing the data stored by the application storage in persistent storage, such as IndexedDB, we propose encrypting the data using the WebAuthn PRF extension described in section 4.3.1. At the time of writing, this extension to the WebAuthn Standard is only available in the Google Chrome web browser.

[5]https://signal.org/docs/specifications/doubleratchet/ (last visited 2024-08-28)

We implemented a getEncryptionKey function that runs a string generated when the application is first run through the WebAuthn PRF. This string can then be used to encrypt the data stored in the IndexedDB. The implementation of this function is shown in Listing 6.8. The function returns a base64 encoded string that can be used as an encryption key.

Implementing Applink with this enabled by default could be annoying for users because this requires users to tap their security key every time they load or reload the application. Furthermore, it requires a security key to use the application.

Therefore, we propose to make this an opt-in feature that users can enable in the settings of the application once the API is implemented in more web browsers.

Listing 6.8: Create a PRF credential on a security key

```
async function getEncryptionKey(){
    if(!localStorage.getItem('encryptionKey')){
        const keyBuffer = new Uint8Array(32);
        crypto.getRandomValues(keyBuffer);
        localStorage.setItem('encryptionKey',
            btoa(String.fromCharCode.apply(null, keyBuffer)));
        await navigator.credentials.create({
            publicKey: {
                rp: {name: "privdb"},
                user: {
                    id: new Uint8Array(16),
                    name: "anonymous@privdb.org",
                    displayName: "Anonymous"
                },
                pubKeyCredParams: [{type: "public-key", alg: -7}],
                timeout: 60000,
                authenticatorSelection: {
                    authenticatorAttachment: "cross-platform",
                    residentKey: "required",
                },
                extensions: {prf: {}},

                // unused without attestation so a dummy value is fine.
                challenge: new Uint8Array([0]).buffer,
            }
        });
    }

    const c = await navigator.credentials.get({
        publicKey: {
            timeout: 60000,
            challenge: new Uint8Array([
                /* does not matter for this use-case */
                1,2,3,4,
            ]).buffer,
            extensions: {
                prf: {
                    eval: {first: new TextEncoder()
                .encode(localStorage.getItem('encryptionKey'))}
                }
            },
        },
    });
    return btoa(String.fromCharCode.apply(null, new Uint8Array(
                c.getClientExtensionResults().prf.results.first)));

}
```

6.11 Anonymity

As we described in section 4.4, sender-recipient unlinkability can be achieved by using an onion routing network. Because of the requirement of onion routing that onion routers are not conspiring, relying strictly on the DHT for discovering onion routers is not feasible. Even though our framework provides some level of confidence in the authenticity of the peers in the DHT network through signed time-based addresses, one can either only pick trusted peers as onion routers, which would considerably reduce the anonymity set, or pick peers at random, risking connecting to a malicious peer. Therefore, a set of trusted onion routers maintained by a trusted entity is required.

Given this constraint, it makes sense to take advantage of an existing anonymity system, such as Tor, to benefit from and further strengthen the existing anonymity set. Tor has a Rust implementation called Arti[6] with an open issue to enable the WASM target[7]. This would allow for the implementation of a bridge to the Tor network accessible using only a web browser.

However, the timeframe for the implementation of the WASM compiler compatibility is uncertain. It requires, among other changes, removing the dependency on TCP sockets in the codebase. This is a non-trivial task, as the codebase is designed to work with TCP sockets.

In this chapter, we have described our proposed system architecture for Applink and explained the reasoning behind our design decisions. In the next chapter, we will evaluate Applink in terms of security, performance, generalizability, and usability.

[6]https://gitlab.torproject.org/tpo/core/arti (last visited 2024-08-28)
[7]https://gitlab.torproject.org/tpo/core/arti/-/issues/103 (last visited 2024-08-28)

CHAPTER 7

Evaluation

In this chapter, we discuss the prototype implementation of the three sample applications defined in section 5.1. We evaluate the functionality, security, privacy, and performance of Applink based on these applications. Additionally, we assess the performance and privacy of Applink in a simulated environment.

In section 7.1, we discuss general remarks about the functionality of Applink and possible techniques for building applications with it. We then discuss the security and privacy of applications built with Applink in section 7.2. In section 7.3, we examine Applink's performance. We evaluate the developer usability of Applink when building applications in section 7.4. Our evaluation of the end-user usability of applications built with Applink is discussed in section 7.5. We discuss the generalizability of Applink in section 7.6. In section 7.7, we discuss the results of the research questions we defined in section 1.2. Finally, we discuss the limitations of Applink in section 7.8.

7.1 Functionality

Applink facilitates communication between peers who know each other's addresses and are online simultaneously. The requirement for peers to be online at the same time to synchronize state is restrictive in Applink's applicability. Currently, web browsers do not allow ServiceWorkers to access the WebRTC API, which would enable background synchronization of application state at the cost of power consumption. A workaround for this is to store the definitive state for every user on a peer that is always online. We envision an extension of this framework capable of running on a Raspberry Pi or a similar low-power, stationary device, serving as the definitive source of truth for a user, akin to a federated server architecture. The "installation" of such applications on the Raspberry Pi could be managed through a separate application installed on the device. However, this approach may hinder widespread adoption due to the technical setup required.

Applications suitable for implementation using a Conflict-Free Replicated Data Type (CRDT) [91] are well-matched for Applink. We have developed a rudimentary Connection Provider for the Yjs CRDT framework [92]. With this integration, any Yjs document can be shared among a designated user group without the need for passwords, while maintaining confidentiality and integrity. This also allows the development of applications that are fully available offline and resynchronize when reconnecting to the internet.

Another design pattern that fits well with Applink is the event-sourcing pattern [93]. We can view CRDTs as a specialized version of event-sourcing: The updates to the document are the events and the document is the state. As Applink has demonstrated its ability to synchronize CRDTs, it is possible to extend Applink to support event-sourcing. This would require a mechanism to determine the missed events of a peer, such as a merkle tree of events similar to most blockchain implementations.

As WebRTC allows connections between pages of different origins, Applink enables applications to communicate directly. Since all messages are authenticated based on the userId, which is unique per origin and web browser, this is not a security problem, as incoming messages are authenticated against and authorized based on the userId. Furthermore, sharing a userId and address with another application strengthens the DHT overlay network because the DHT layer treats peers running different applications the same.

However, this poses a usability challenge. For an application to establish a connection between peers, the address must be exchanged out of band. This can be done via a link that can be shared. Establishing a connection between different applications of the same user can be tricky. In the case of the invoicing application and the BCS, we chose to copy the address to the clipboard. This is a suboptimal solution, as pasting this address into the wrong application could lead to spam invoices being deposited into the invoice tracking system.

One limitation of Applink is that it only allows one web browser tab to be open at a time. This is because Applink performs an IndexedDB database upgrade when the page is loaded to ensure the database schema is up to date. A possible workaround is to display a placeholder page until a database connection is established, informing the user that only one instance of the application can be open at a time.

For administrators hosting applications developed with Applink, the savings on hosting costs are significant. Applink requires only a static server to host the source code and a push server to relay messages. All the user data is stored on the user's device, reducing hosting costs for user-generated content.

7.2 Security and Privacy

In this section, we evaluate the security and privacy of applications built with Applink.

7.2.1 Security

As Applink allows arbitrary users to invoke calls, applications are at risk of novel vulnerabilities similar to those found in smart contracts on the Ethereum blockchain [94], such as broken access control or arithmetic bugs. For instance, an application author might forget to validate the origin of a function call. Consider the code snippet in Listing 7.1, which evaluates to true despite the two numbers not being equal.

Listing 7.1: JavaScript integer overflow

```
Number.MAX_SAFE_INTEGER + 100 == Number.MAX_SAFE_INTEGER + 101
output: true
```

There are multiple libraries for input validation in JavaScript, such as validatorjs[1] and is2[2]. Application authors can utilize these libraries to validate input and prevent such bugs. Because the controller class of the MVC pattern can be the same class that allows RPC calls, the application author can spot inconsistencies in input validation more easily.

Due to the implicit public-key-based authentication, users are not required to remember passwords or use password managers.

We designed the DHT protocol to not respond with a large payload to requests with a small payload to avoid request amplification attacks. Moreover, Applink establishes an implicit web of trust through the distribution of push addresses. Without knowing at least one peer in a trusted overlay network, an attacker cannot connect to it. Even if a WebRTC connection is established, messages will not be processed unless a contact establishment handshake via Push has occurred beforehand, which requires knowing the full contact information of the peer. This significantly reduces the attack surface for an attacker who is not a contact of the victim.

Claim: A peer will only process a message if it is from a known contact.
Security Argument:

(0) Consider the `_r_onMessage` function from Listing 7.2 of the `DhtTransport` class. This is the only function in that class that invokes the message listeners.

(1) The function call to `contactManager.getAddressFromTimeBasedAddress` will throw an error if the time-based address does not belong to a contact. Therefore, a message will only be processed if the time-based address belongs to a known contact.

(2) Since the message is from a contact, and it has been decrypted using RSA-OAEP-AES, the sender must know the private key of the RSA key pair with which the message was encrypted.

(3) All calls to `rsaOaepeAesMessage.addContact` are preceded by a call to `contactManager.verify`, which checks the signature of the RSA-OAEP-AES public key against the RSA-PSS-SHA256 public key associated with the time-based address if the sender is a contact, which is confirmed by (1).

[1] https://www.npmjs.com/package/validatorjs
[2] https://www.npmjs.com/package/is2

(4) Producing this signature requires the private key of the RSA-PSS-SHA256 key pair. Therefore, we infer that the sender, who has access to the private key of the RSA-OAEP-AES key pair, also knows the private key of the RSA-PSS-SHA256 key pair.

(5) Because the RSA-PSS key pair signed the RSA-OAEP-AES key pair (as noted in 4) and is associated with the time-based address (as noted in 3), we can conclude that the RSA-OAEP-AES key pair belongs to a known contact.

(6) Given (2) and (5), we can conclude that the message is indeed from a known contact.

(7) From (0), (1), and (6), we can finally conclude that a message will only be processed if it is from a known contact.

Listing 7.2: _r_onMessage function of the DhtTransport class

```
async _r_onMessage(from: string,
    { message, to, ttl }: DhtParameterMap['send']) {

    if (to === await this.contactManager.getOwnTimeBasedAddress()) {
        message =
            (await this.rsaOaepeAesMessage.decryptMessage(message))
            .payload;

        /* if this message can be interpreted
           by the longAsyncRF of the DhtTransport, do so */
        if (!(await this.longAsyncRF.messageRecieved(from, message))) {
            /*otherwise, pass it to the listeners */

            /* resolve time based address
               (only works for contacts, will throw if not a contact) */
            from = await this.contactManager
                .getAddressFromTimeBasedAddress(from);
            this.listeners.forEach(listener => listener(from, message));
        }
    } else {
        //if the message is not for us, forward it
        if (ttl > 0) {
            await this.send(to, message, false, from, ttl);
            console.log(`forwarded message from ${from} to ${to}`);
        }
    }
}
```

One potential attack vector for de-anonymizing the time-based address is to perform a handshake with a peer to obtain its public IP address. This can be mitigated by limiting WebRTC traffic to run through TURN servers. However, this channels all traffic through centralized infrastructure, increasing the attack surface and causing a financial burden on the TURN server operator.

Double Onion Circuit Anonymity Security Argument

In section 6.7, we describe the double onion circuit protocol to unlink time-based addresses from static identifiers.

Claim: A peer can unlink their static identifier from their time-based address.
Security Argument:

Assumptions:

(A1) An onion circuit between two peers, Y and Z, guarantees sender-recipient unlinkability between Y and Z, such that no other peer can observe the communication link between Y and Z. Furthermore, only the peer who establishes an onion circuit can identify the link between the hops of the onion circuit.

(A2) The overlay network uses time-based addresses, which are used to route messages within the overlay network.

(A3) To join the overlay network, a peer can use a random, single-use address as a time-based address for the initial connection.

Steps:

(1) Two peers, A and B, have established a direct WebRTC connection through web push and therefore know each other's static push identifiers.

(2) Peer B establishes an onion circuit, BC, to peer C and tunnels all of A's traffic through C. This is done in such a way that no peer other than B can relate the traffic exiting C back to B.

(3) Due to (2), peer A and peer C are unable to observe the origin of the onion circuit BC, thereby preventing the linking of B's static identifier (push registration) to B's time-based address.

(4) Peer A establishes an onion circuit, AD, to peer D and tunnels all of their traffic through it. This is done in such a way that no peer other than A can relate the traffic exiting D back to A.

(5) Due to (2), the onion circuit AD is routed through the onion circuit BC.

(6) Due to (A1) and (2), even if peer B does not establish an onion circuit, A's exit node is D, and B cannot correlate any traffic exiting D with A.

(7) Due to (6), A can use their time-based address to establish a direct WebRTC connection with any peer on the overlay network using the routing of the overlay network, without any other peer being able to link this traffic to A's static identifier.

Given (3) and (7), a peer can unlink their static identifier from their time-based address.

ProVerif Verification

We have verified the unlinkability of addresses over time, as well as the unlinkability of time-based addresses, using ProVerif [95]. The ProVerif code is shown in Listing 7.3.

Listing 7.3: Proverif code for unlinkability of time-based addresses and message confidentiality

```
free c:channel.
type key.

fun senc(bitstring,key):bitstring.
reduc forall m:bitstring, k:key; sdec(senc(m,k),k) = m.

type skey.
type pkey.

type address.
type tbaddress.


fun pk(skey):pkey.
fun aenc(bitstring, pkey):bitstring.

reduc forall m:bitstring, k:skey; adec(aenc(m, pk(k)), k) = m.

fun sign(bitstring, skey):bitstring.

reduc forall m:bitstring, k:skey; checksign(sign(m,k), pk(k)) = m.

fun signPk(pkey, skey):bitstring.

reduc forall m:pkey, k:skey; checksignPk(signPk(m,k), pk(k)) = m.


fun timeBasedAddress(address, bitstring):tbaddress.

reduc forall a:address, ts:bitstring; timeBasedAddressDecode(timeBasedAddress(a, ts), a, ts

fun associate(address, pkey):bitstring.

reduc forall a:address, pubk:pkey; getPkByAddress(associate(a, pubk), a) = pubk.


free Message:bitstring [private].
free Timestamp:bitstring.
free AddrB:address [private].

event sent(bitstring).
event recieved(bitstring).

let Alice(skA:skey, ltPkB:pkey, addrB:address) =
    in(c, pkBMsg:bitstring);
    let pkB = checksignPk(pkBMsg, ltPkB) in
    let ass = associate(addrB, pkB) in
    in(c, tbaB:tbaddress);
    in(c, ms:bitstring);
    let addrM = timeBasedAddressDecode(tbaB, addrB, Timestamp) in
    let pkM = getPkByAddress(ass, addrM) in
    let me = checksign(ms, pkM) in
    let m = adec(me, skA) in
    event recieved(m);
    0.

let Bob(ltskB:skey, pkA:pkey, addrB:address, m:bitstring) =
    (* create a fresh keypair, sign the public key with the long term signature key and publish i
    new skB:skey;
    out(c, signPk(pk(skB), ltskB));
    out(c, timeBasedAddress(addrB, Timestamp));
    (* encrypt the message with the public key of the other party and sign it with the fresh key
    let me = aenc(m, pkA) in
    let ms = sign(me, skB) in
    event sent(m);
    out(c, pk(skB));
    out(c, ms);
    0.

query attacker(Message).
query attacker(AddrB).

query x1:bitstring; event(recieved(x1)) ==> event(sent(x1)).

process
    new ltskA:skey;
    new ltskB:skey;
    ((!Alice(ltskA, pk(ltskB), AddrB) | !Bob(ltskB, pk(ltskA), AddrB, Message)) )
```

OWASP Top 10

Next, we will discuss the OWASP top 10[3] vulnerabilities and their applicability to Applink.

(1) Broken Access Control: Access control is application-specific. Applink, particularly through the `RPCInterface` interface, encourages developers to authenticate and authorize every function call, as the first parameter of every remotely callable function is the userId of the caller.

(2) Cryptographic Failures: Applink's reliance on the WebCrypto API, rather than requiring application developers to implement cryptographic functionality, reduces the likelihood of cryptographic failures.

(3) Injection: Cross-site scripting remains a threat to applications using this framework. Modern templating frameworks like Lit partially mitigate this risk by automatically escaping user input[4].

(4) Insecure Design: Applink, especially with its `RPCInterface` abstraction, aims to clarify which functions can be invoked by any remote peer. It facilitates local testing and

[3] `https://owasp.org/Top10/A00_2021_Introduction/` (last visited 2024-08-28)
[4] `https://github.com/lit/lit.dev/issues/448`

simplifies code reasoning. However, centralized applications, which can monitor activities more easily and store all data in a central database, reduce the mental load for developers associated with partitions.

However, developers could accidentally break some of the anonymity features of Applink, for example by sharing user IDs. Developers could even completely break the anonymity benefits of Applink by storing all data in a central database, adding tracking scripts such as Google Analytics[5], including monitoring scripts such as Rollbar[6] or Sentry[7], or by using a centralized authentication service such as Auth0[8].

(5) Security Misconfiguration: Applink's reduction of dependency on a properly functioning server decreases the attack surface. However, security headers such as HTTP Strict Transport Security (HSTS) and Content Security Policy (CSP) should be set by the application developer/administrator.

(6) Vulnerable and Outdated Components: As highlighted by Zahan et al. [96], supply chain vulnerabilities in the JavaScript and NodeJS ecosystems are a growing concern. This framework does not directly address this issue.

(7) Identification and Authentication Failures: Applink's requirement for users to exchange initial connection information out of band makes it vulnerable to information leakage to attackers. Ideally, Applink would store the address in the DHT and exchange the DHT address and a decryption key out of band, deleting the DHT entry after establishing the connection.

(8) Software and Data Integrity Failures: Developers should ideally be able to sign their applications for web browser-independent validation, allowing for integrity protection even when multiple servers serve application code due to censorship events. Validating any signature in JavaScript would be ineffective, as attackers could modify the validation code. A possible solution involves using a web browser extension for validation, but this contradicts the initial assumption that web browser extensions cannot be relied upon, especially in mobile web browsers.

(9) Security Logging and Monitoring Failures: Applink's lack of support for centralized logging supports privacy but potentially detracts from security. Proper pre-deployment application testing is essential for security. A possible approach involves defining a set of events that can be reported to a centralized server managed by a trusted entity, although this does not fully resolve the logging issue.

(10) Server-Side Request Forgery: Applink is not vulnerable to server-side request forgery, as it only uses a server to relay push messages.

[5]`https://analytics.google.com` (last visited 2024-08-28)
[6]`https://rollbar.com` (last visited 2024-08-28)
[7]`https://sentry.io`
[8]`https://auth0.com/` (last visited 2024-08-28)

Censorship Resistance

Applications utilizing Applink can be easily deployed to a web server and accessed via a QR code, providing a straightforward distribution method. Furthermore, even if the web server is blocked, the ServiceWorker API allows for offline storage of all application code, enabling continued application use.

The web server, which removes the CORS header for the push service, can also be easily deployed and distributed automatically to trusted contacts.

Blocking the push service or WebRTC is impractical, as they are widely used, and protocols cannot be blocked on an application-by-application basis. Applications built with Applink are as usable as regular websites, making them accessible to computer novices.

Social Availability

Because applications built upon this framework run in a web browser and are similar to other web applications, it would be hard to stigmatize usage of those applications the same way some organizations have attempted to stigmatize usage of the Tor browser, such as this page run by the Austrian government quoting Josef Pichlmayr advising people that:

> Es gibt eigentlich keinen Grund, im Darknet zu sein. Außer man ist Journalistin oder Journalist, studiert Cybersicherheit oder macht soziologische Studien. Ansonsten hat man dort nichts verloren. Dinge, die im Clear Web nicht erhältlich sind, gibt es dort aus guten Gründen nicht.
>
> ---
>
> English translation produced with GPT-4o:
>
> There is actually no reason to be on the dark web. Unless you are a journalist, studying cybersecurity, or conducting sociological studies, you have no business there. Things that are not available on the clear web are not available there for good reasons.

[a].

[a]Josef Pichlmayr interviewed by `https://www.onlinesicherheit.gv.at/Services/News/Darknet-Ueberblick.html` (last visited 2024-08-28)

Or attempts by the European Union to undermine privacy law and human rights by deploying mass surveillance technology under the pretense of preventing child abuse [97][9].

Given the ease of deployment and the option to transfer all data to a different domain that our framework provides, it would be difficult to stigmatize or block applications built with Applink faster than they can be redeployed.

[9]`https://edri.org/our-work/how-a-hollywood-star-lobbies-the-eu-for-more-surveillance/` (last visited 2024-08-28)

7.2.2 Privacy

Because Applink does not store user data on centralized infrastructure and does not require users to sign up, it enhances user privacy over centralized applications.

Transformer-based language models can analyze user data [98] as well as retrieve data matching search queries, especially in edge cases where exact keywords are hard to find, such as searching for specific user records [99].

Therefore, it is crucial to limit the amount of data accessible to any single entity, such as data stored unencrypted on centralized infrastructure.

As described in section 4.4, Applink could be extended to support sender-recipient unlinkability through onion routing. However, implementing onion routing entails years of development, testing, and user adoption. A more viable approach might be to utilize the Tor onion routing network, which the project is currently exploring but has not yet implemented.

It should be noted that applications using this framework can still implement tracking mechanisms alongside it. JavaScript can initiate arbitrary connections to any server, and Applink does not prevent this.

7.2.3 Threat Model

As outlined in the threat model in subsection 5.2.2, we identify three types of adversaries: *(1) Censors are limited in their ability to shut down applications using Applink.*

Applink can utilize a list of proxies to send messages to the push infrastructure. These proxies can be automatically distributed through applications without user intervention, making it difficult to block messages sent to push infrastructure.

We assume the delivery of messages by the push infrastructure will not be shut down. Therefore, we only concern ourselves with solving for the path from the client to the push infrastructure.

Even if an application's domain is blocked, users can still access the application if it employs web browser caching via the Service Worker API. Furthermore, the entire application state can be automatically transferred to an identical application hosted on a different domain.

As the process of blocking IP addresses or domains for censorship purposes is slow, a dedicated community can outpace a censor. This was exemplified by the Tor project's experience in Russia, where lists of IP addresses were compiled into Excel spreadsheets and emailed to the censors[10]. The servers proxying the push traffic can also require a pre-shared key and be distributed only to a trusted set of contacts to prevent active probing of the infrastructure.

[10] `https://youtu.be/g5ZiBYR-1MM?si=JygB7VziJ21QM06S&t=1224` (last visited 2024-08-28)

(2) Malicious administrators cannot prevent individual users from accessing or using an application built on Applink. However, they can introduce malicious code into a website. One possible solution to this problem is to monitor the code of an application for changes. This could be in the form of a third-party website where users sign up to receive a notification if malicious code is detected.

Because detecting malicious code is difficult to automate, this would require human review. There might be the possibility of automating this review using machine learning, specifically transformer-based pre-trained language models.

(3) Malicious users can attempt to send malicious messages to other users that exploit vulnerabilities in the application or framework. Given that Applink has not been extensively reviewed, we must assume some vulnerabilities might eventually be discovered. Applications are encouraged to authenticate a request as the first step in its invocation, meaning a malicious user must first be added as a contact to launch an attack against more complex and, therefore, more error-prone parts of an application.

Applications should be tested with unexpected data types (e.g., arrays or objects instead of strings) in their parameters to anticipate potential attack scenarios. One aid in improving application security could be to create a fuzzer for `RPCInterface` that automates testing for common error causes.

Malicious users can only determine that communication between peers is happening if they know both addresses (to correlate time-based addresses) and if the user is part of the path between these peers. Since every peer changes their time-based address regularly and at the same time, correlating communication sessions and consistently being part of the path between peers is made more difficult as the address in the DHT is determined by a secure hash function and a connection through a web-of-trust is required to establish a connection. This is because joining the DHT requires knowing the push registration information of a peer that is already part of the DHT.

7.2.4 Confidentiality

Assuming a correct application implementation, all data is stored by Applink on the user's devices and transmitted only to authenticated and authorized users. No central authority can read or modify the data of applications.

However, social engineering attacks on individual users can lead to attackers gaining access to the information of a user. Application authors should keep this in mind and design their applications to minimize the impact of such attacks.

7.2.5 Simulation of Malicious Honest-but-Curious Attacker Peers

We extended the simulation described in section 6.5 to include malicious peers that are honest-but-curious. These peers count the packets they receive and send and from whom to whom they are sent. Subsequently, the peers guess which peers are communicating with each other based on the number of packets sent and received. In our simulation,

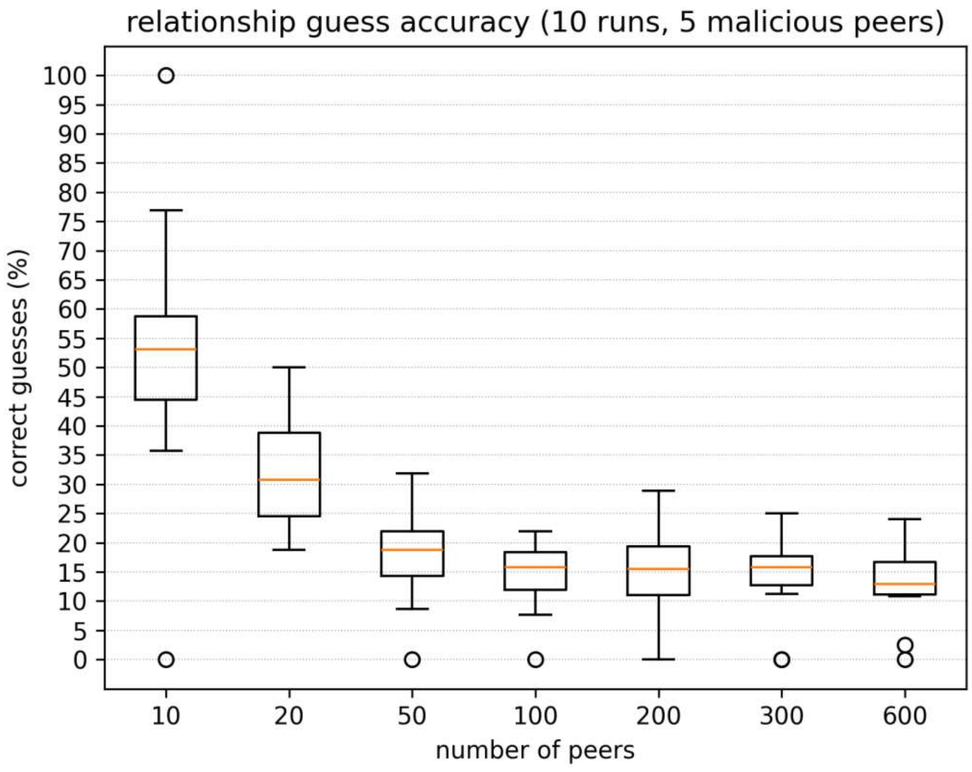


Figure 7.1: Success rate of packet-counting attack

we guess that communication above the mean number of packets sent and received is communication between two peers. We always simulate five malicious peers with a varying number of other peers.

The results are shown in Figure 7.1. The median success rate of the packet counting attack with five malicious and five honest peers is less than 5% better compared to random guessing across all simulated numbers of peers (10, 20, 50, 100, 200, 300, 600). For 600 peers with five malicious peers, we compared random guessing peer relationships to packet counting peer relationships and found that the packet counting attack has a 2% advantage over random guessing as shown in Figure 7.2.

This experiment demonstrates the need for additional measures to provide sender-recipient unlinkability. However, it also demonstrates that time-based addresses can significantly reduce the cross-session tracking capabilities of malicious peers. This reduction in tracking capabilities is due to the difficulty for an attacker to determine if two communicating peers are the same as in the previous time-based address interval. We are ignoring the fact that direct connections via WebRTC reveal the IP address of the peer, as the IP address can be obfuscated by a peer with the help of a VPN.

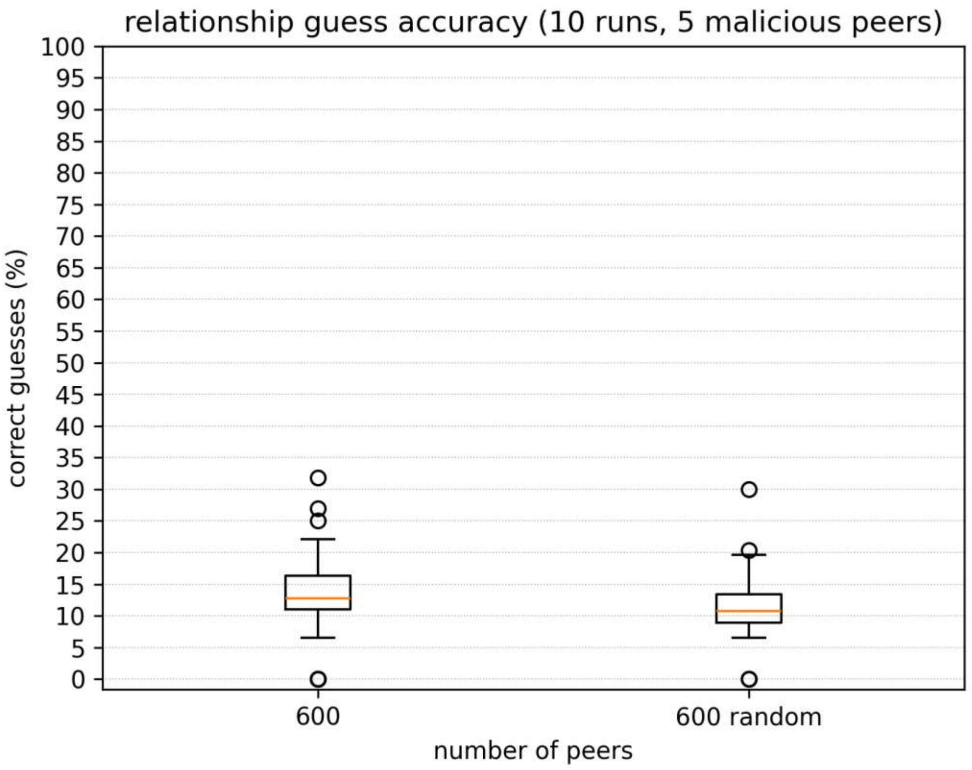


Figure 7.2: Advantage over random guessing through packet counting

7.3 Performance

We evaluated three main performance metrics for Applink: page load time, connection establishment time, and synchronization overhead.

The page load time is the time it takes for the application to be usable after the user navigates to the application's URL.

The connection establishment time is the time it takes for a peer to establish a connection with another peer.

Synchronization overhead is the CPU overhead generated by Applink when synchronizing application state between peers. We evaluated these metrics using the notes application because it is the application that works best with Applink and is the most feature-complete prototype implementation.

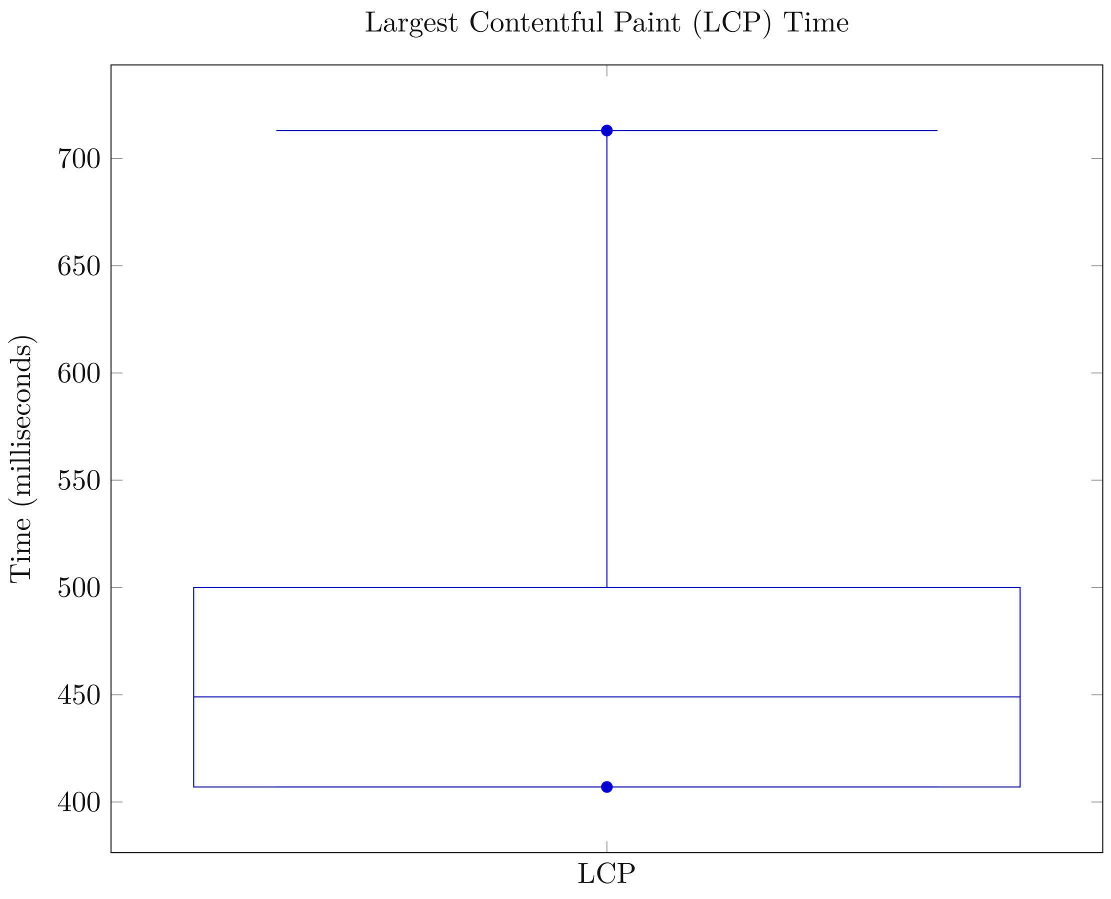


Figure 7.3: Boxplot of Largest Contentful Paint (LCP) for the notes application on Pixel 6 Pro Running Chrome 121.0.6167.178 on Android 14.

7.3.1 Page Load and Bootstrapping

We measured the Largest Contentful Paint (LCP)[11] metric using Chrome DevTools, as seen in Figure 7.4, for the notes app to assess page load performance. In the case of the notes application, the LCP event fires when the user's notes are visible on the screen.

The measurements, conducted on a Pixel 6 Pro running Chrome 121.0.6167.178 on Android 14 and served over a local network, show a median LCP of 449ms, with the 25th percentile at 407ms and the 75th percentile at 500ms, and the highest outlier at 713ms (N=20). These results meet Applink's performance requirements. It is important to note that this measure is independent of network latency, as all application data can be loaded from web browser storage. A boxplot of the LCP times is shown in Figure 7.3.

[11]`https://web.dev/articles/lcp#what-is-lcp` (last visited 2024-08-28)

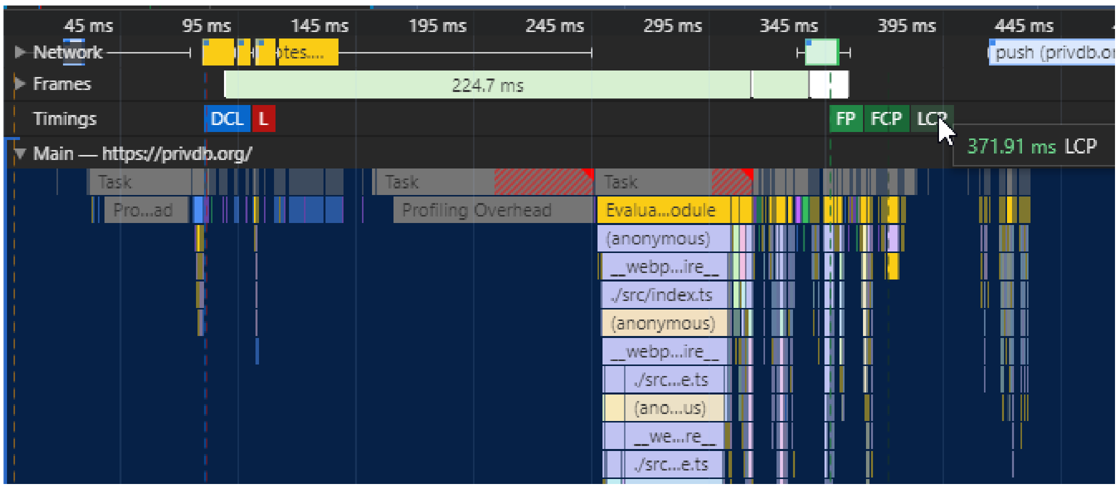


Figure 7.4: Page load time of the notes application on a Pixel 6 Pro running Chrome 121.0.6167.178 on Android 14.

7.3.2 Connection Establishment

The bootstrapping process requires the exchange of two push notifications between peers containing the WebRTC offer and answer.

Our initial assumption was that the bottleneck of this process would be the push notification delivery time. While this is true for most cases, ICE candidate gathering can also be a bottleneck.

In some circumstances, the bootstrapping can be unacceptably slow (40-50 seconds on a Pixel 6 Pro running Chrome 121.0.6167.143 connecting to a desktop PC running Chrome 121.0.6167.140, with both devices on the same local network), as gathering the ICE candidates for WebRTC encounters a 40-second timeout if a STUN server cannot be reached[12].

This timeout occurred with the Pixel 6 Pro but not with the desktop machine in our testing setup due to only IPv4 connectivity being available. On the desktop machine, the median connection time is 1,446ms, with the 25th percentile of 1,331ms and a 75th percentile of 7,972ms (N=20). The highest outlier was 15,820ms. This high span is caused by a race condition when two peers attempt to connect to each other at the same time. If both peers enter the state where they start a handshake, they will wait for a random timeout before trying again. Once a connection is established, Android devices running Google Chrome keep the website running in the background, allowing for limited synchronization while the application is not open. By shortening the ice-gathering stage of the WebRTC connection, as shown in Listing 7.4, connection time on Android devices is reduced to 10-20 seconds. With the fix applied, we measured the connection times as seen in Figure 7.5.

[12] `https://stackoverflow.com/questions/76491417` (last visited 2024-08-28)

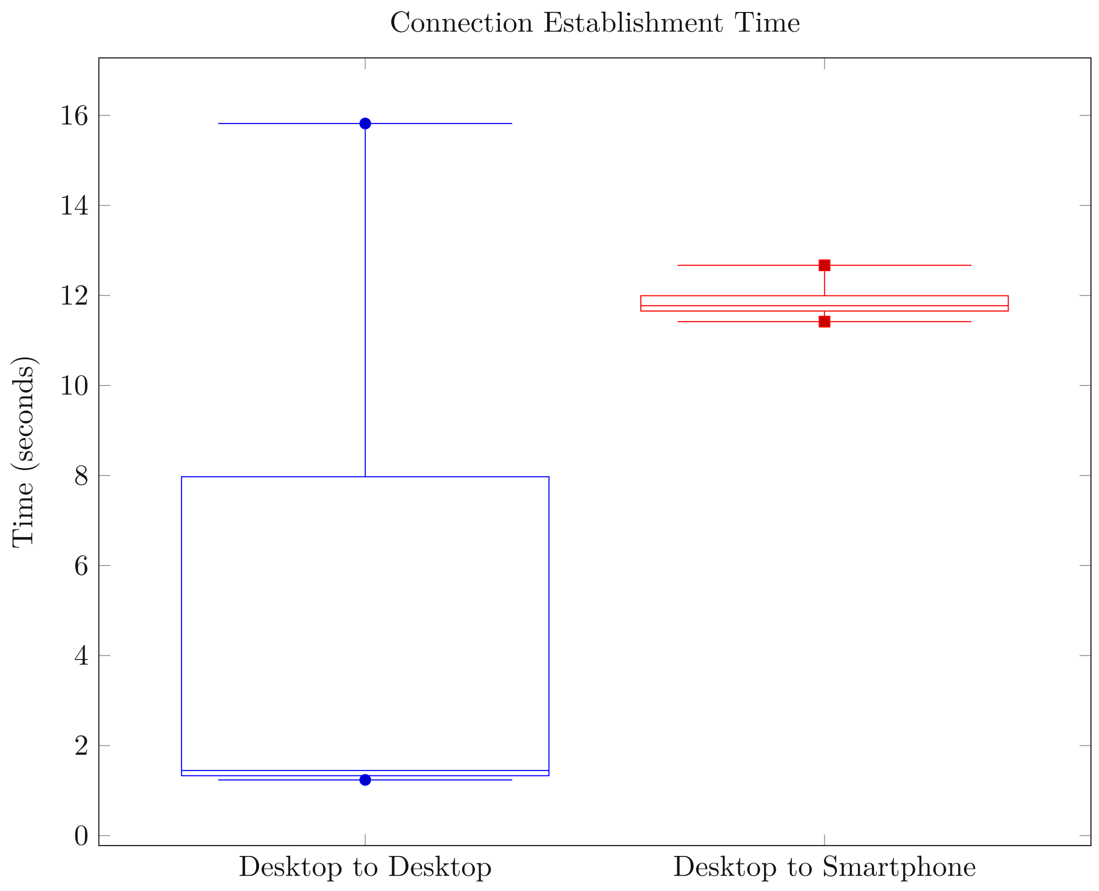


Figure 7.5: Boxplot of Time to first connection established for the notes application.

We evaluated how long a Pixel 6 Pro running Chrome 121.0.6167.178 on Android 14 keeps the notes app running in the background and found it to be 5 minutes when plugged into power and 3 minutes when unplugged, with battery saver mode turned off.

Listing 7.4: Shortcut for the ice gathering stage of the WebRTC connection

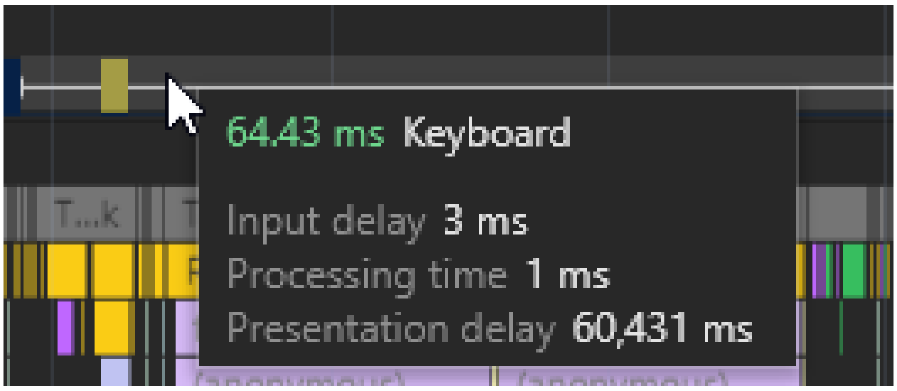
```
const icePromise = new Promise<string>(resolve => {
    connection.onicegatheringstatechange = () => {
        if (connection?.iceGatheringState === 'complete') {
            resolve(JSON.stringify({ candidates, offer }));
        }
    };
});
return await Promise.race<string>([icePromise,
    (async () => {
        await timeout(10000);
        if (candidates.length > 0) {
            return JSON.stringify({ candidates, offer });
        }
        return icePromise;
    })()
]);
```

7.3.3 Synchronization

Chrome DevTools indicates that synchronizing the application state during user interaction does not noticeably affect the application's performance. As demonstrated in Figure 7.6, the notes application's input events have a 60ms presentation delay. This is within our target of 100ms for responding to user input. The presentation delay occurs because most of the encryption work is handled by the WebCrypto API, which operates in a separate thread, preventing the web browser from blocking while rendering a frame. The notes application uses libsignal's JavaScript implementation of curve25519, as it is not yet supported by the WebCrypto API[13]. Performance could be further improved by moving the curve25519 implementation to a web worker. The execution time of curve25519 in JavaScript is shown in Figure 7.7. The execution time is less than 10ms and, therefore, within our performance requirements defined in subsection 5.2.4.

We used the simulation described in section 6.5 to evaluate the expected success rate of message delivery without relying on storing data on a central server or on peers that are not involved in a given data exchange. For this, we assumed that peers' interaction frequencies follow a Zipfian distribution [100] and used the zipfian npm package[14] to generate the distribution. Our simulation uses discrete time steps ("ticks"). For every tick, every peer has a 1% chance to go online or offline. Online peers that have established at least one connection attempt to send five messages to their contacts per tick according to their fixed contact frequency distribution determined by the Zipfian distribution. Online peers attempt to connect to one additional peer per tick using the logic of the k-bucket DHT library[15]. We define the success rate as the percentage of messages that can be delivered in the DHT network at that point in time. The simulation is run for

[13]`https://github.com/w3c/webcrypto/pull/362` (last visited 2024-08-28)
[14]`https://github.com/willscott/zipfian` (last visited 2024-08-28)
[15]`https://www.npmjs.com/package/k-bucket` (last visited 2024-08-28)



64.43 ms Keyboard

Input delay 3 ms
Processing time 1 ms
Presentation delay 60,431 ms

Figure 7.6: Synchronization of the notes application on a Pixel 6 Pro.

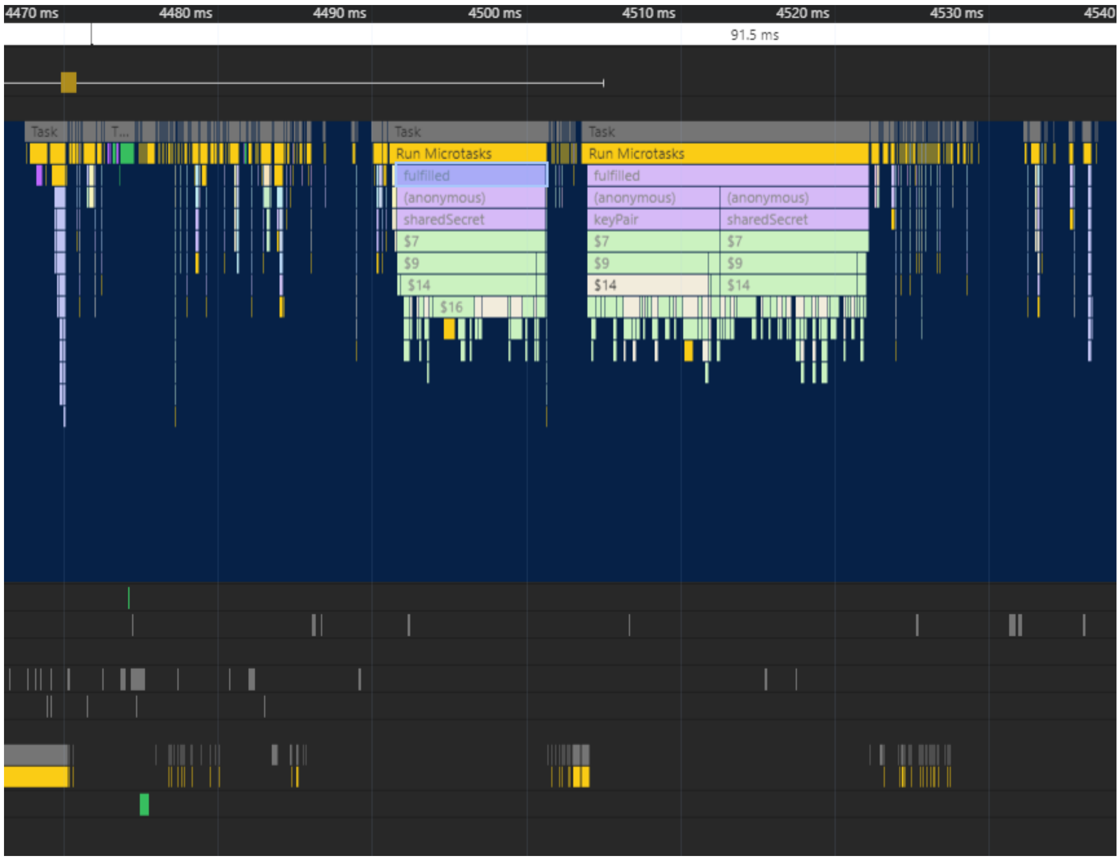


Figure 7.7: Execution of curve25519 in JavaScript on a Pixel 6 Pro.

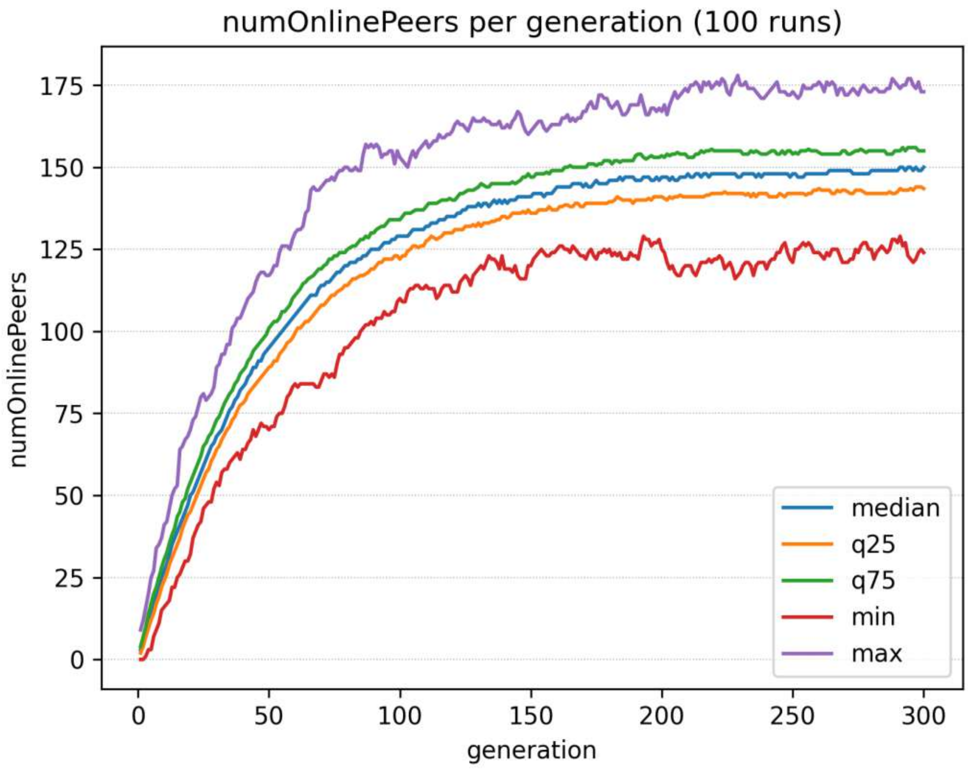
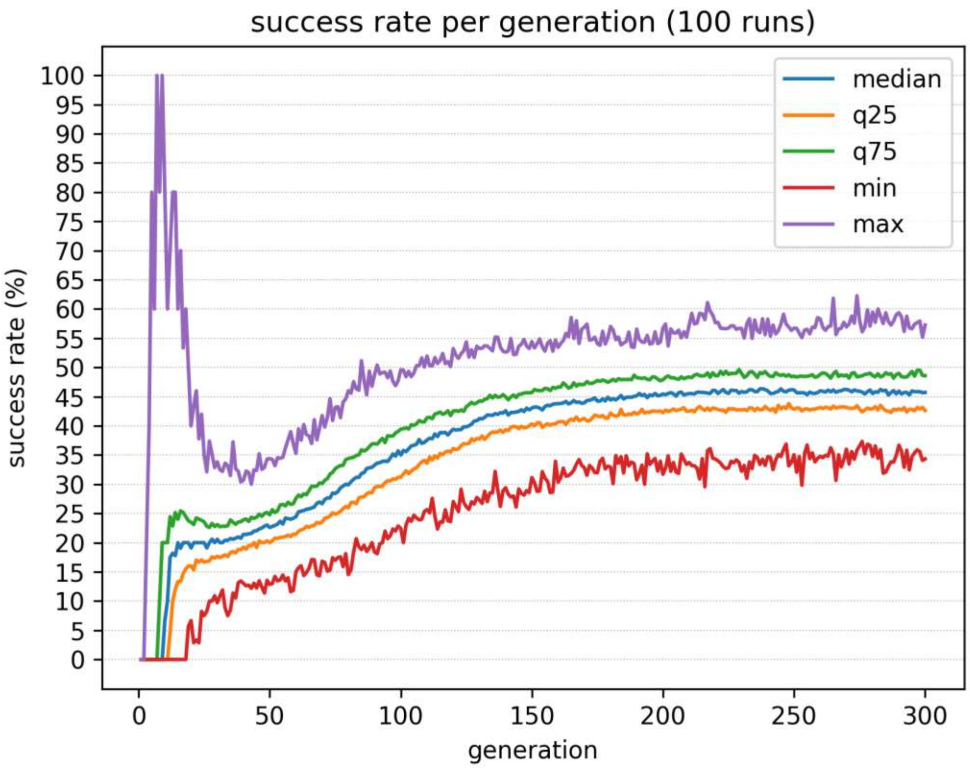


Figure 7.8: Number of online peers over time

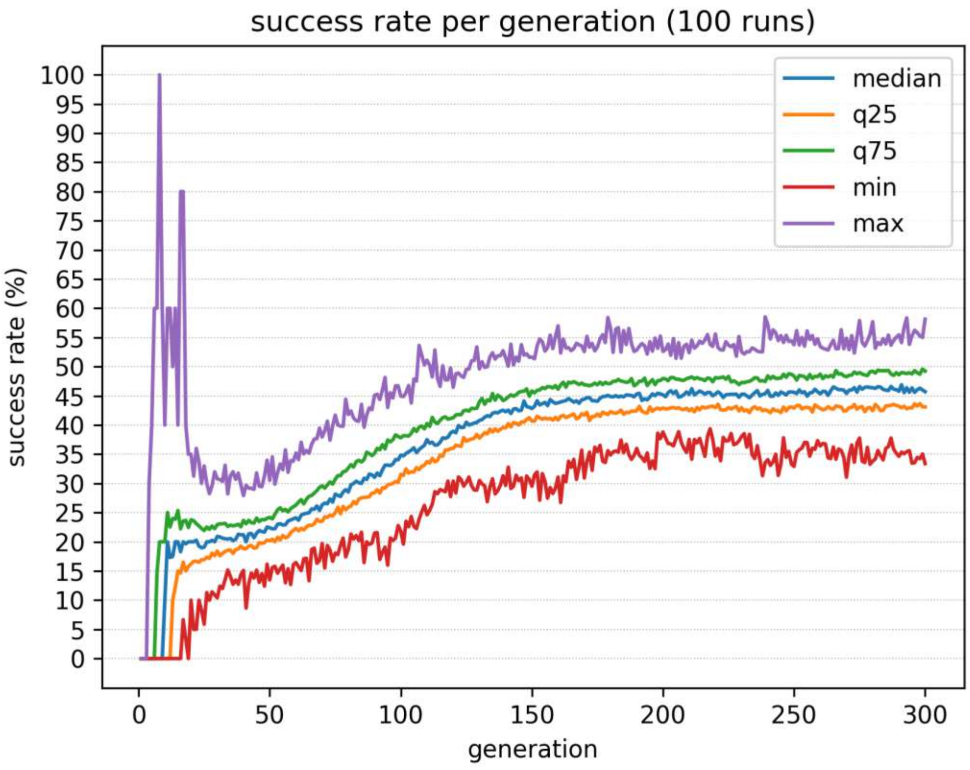
300 ticks with 300 peers. The results are shown in Figure 7.9a. As the number of online peers approaches 50% of all peers, the success rate in the median is about 45%. The number of online peers over time is shown in Figure 7.9a. We ran the simulation with a uniform contact distribution instead of the Zipfian distribution, and the results are shown in Figure 7.9b. There is no significant difference in the success rate of message delivery between the two distributions. This shows that the Zipfian distribution of contact frequencies does not have a significant impact on the success rate of message delivery. The discrepancy between the number of online peers and the success rate of message delivery is caused by our simplified simulation code for bootstrapping peers.

In the real world, it is likely that peers that want to communicate with each other are online at the same time, at least in some use cases such as collaborative editing of a document while in a meeting or sending instant messages. This would increase the success rate of message delivery.

The code simulating the DHT network routing is shown in Listing 7.5. We set the time-to-live of a packet to 20. If the current peer is not online or the time-to-live is zero, the packet is considered failed. If the packet reaches its destination, the packet



(a) Number of successful messages delivered



(b) Number of successful messages delivered with uniform contact frequency distribution

Figure 7.9: Simulation of the DHT network with 300 peers

is considered successful. Our routing logic allows packets to not be sent to the nearest contact if the packet came from that contact, as this would simply create a loop until the time-to-live hit zero. This can happen if the closest contact is still in their bootstrapping phase or when the recipient of a packet is offline.

We track the observed traffic per peer to analyze the viability of a packet counting attack as described in subsection 7.2.5.

Listing 7.5: sendPacket function of the simulation

```
/**
 *
 * @param {Peer} from
 * @param {Peer} to
 * @returns
 */
sendPacket(from, to, ttl = 20) {
    if (!this.isOnline) {
        this.failedPackets++;
        return;
    }
    if (!this.observedTraffic.has(from.timeBasedAddress)) {
        this.observedTraffic.set(from.timeBasedAddress, new Map());
    }
    if (
        !this.observedTraffic
        .get(from.timeBasedAddress).has(to.timeBasedAddress)
       ) {

        this.observedTraffic.get(from.timeBasedAddress)
          .set(to.timeBasedAddress, 0);
    }
    this.observedTraffic.get(from.timeBasedAddress)
        .set(to.timeBasedAddress,
          this.observedTraffic.get(from.timeBasedAddress)
            .get(to.timeBasedAddress) + 1);

    if (to === this) {
        this.successfulPackets++;
        return;
    }
    if (ttl <= 0) {
        this.failedPackets++;
        return;
    }
    const closestEntries =
      this.kbucket.closest(to.encodedTimeBasedAddress, 3);

    for (const contactKEntry of closestEntries) {
        const contact = this.connections
          .find(c => c.encodedTimeBasedAddress === contactKEntry.id);

        if (contact === from && closestEntries.length > 1) {
            continue;
        }
        if (contact) {
            contact.sendPacket(this, to, ttl - 1);
            return;
        } else {
            throw new Error('Contact not found');
        }
    }
```

```
        this.failedPackets++;
    }
```

7.4 Developer Usability

Applink can save developers time to deal with authentication on the backend, as applications built with Applink run entirely in the web browser. Additionally, Applink provides a built-in contact system that can be further developed. The two components of the Notes application that are not related to the user interface, the NoteShareController and the PrivDbProvider, contain fewer than 400 lines of code. This demonstrates that Applink is suitable for small applications and prototypes. Moreover, the `RPCInterface` abstraction allows for local testing without a web browser, provided that the tested component does not rely on a web browser-only API.

However, exposing functions of an object to be called by anyone without this being clearly marked in the code can lead to mistakes. Applink enforces the naming of functions registered in an `RPCInterface` to contain "_r_" to remind developers that these functions can be called by any remote peer. However, data types are not enforced by Applink in the current implementation. Proper input validation is still the responsibility of the developer and is a potential source of bugs. Therefore, we plan to implement Zod schemas[16] for input validation in the future.

Monitoring a distributed privacy-focused application without compromising the privacy of the users is a challenge. Applink does not provide a solution to this problem.

One possible solution is to ask tech-savvy users to send stack traces from their web browser console to the developer. However, this is very limited as obtaining logs from a mobile web browser is not possible without connecting the device to a computer. It could be near impossible to obtain a stack trace from a not-trivial-to-reproduce bug.

To mitigate this, best practices should be followed during development, such as testing, linting, and code reviews. Furthermore, developers should consider using TypeScript to catch type errors at compile time or build core application logic in Rust and compile it to WebAssembly, which is becoming increasingly common[17,18,19].

Applink fulfills the requirements outlined in subsection 5.2.5. It offers support for building distributed applications with a focus on privacy and security.

Furthermore, Applink allows developers to build transactional flows with the `RPCInterface` abstraction. Especially with the ability to declare an expected callback to be called

[16] `https://zod.dev/` (last visited 2024-08-28)
[17] `https://2021.stateofjs.com/en-US/other-tools/#non_js_languages` (last visited 2024-08-28)
[18] `https://2022.stateofjs.com/en-US/other-tools/#non_js_languages` (last visited 2024-08-28)
[19] `https://2022.stateofjs.com/en-US/features/other-features/` (last visited 2024-08-28)

within a timeframe, as shown in Listing 6.6, Applink allows developers to build robust applications that can recover from network failures.

It also allows for applications to exchange data across origins as described in subsection 5.2.3. API endpoints accessible from other origins can be implemented using the `RPCInterface` abstraction. Because all calls to the `RPCInterface` can come from any other peer, the threat model does not change because of this feature. We ensure that RPC calls can only be invoked from trusted contacts rather than relying on the same-origin policy.

7.5 End-User Usability

For end-users, Applink improves the initial application experience by not requiring a signup to get started using an application. However, this could also lead to user confusion as it does not adhere to the common patterns users are familiar with.

Adding other devices can be implemented to be similar to the process of logging in on a new device in a centralized application such as Discord[20] and Steam[21]. Users can use a QR code to pair a new device with the application. If such a flow is implemented, the application state can be synchronized between the existing and the new device because both applications need to be open to scan the QR code and to present the QR code.

However, users could be confused by the fact that changes do not synchronize across devices unless their devices are online at the same time. This could be worked around on a per-application basis depending on the privacy requirements.

Furthermore, users who are unaware of Applink storing all data on-device might not expect their data to be available offline and therefore not take advantage of this feature, further diminishing the benefits of Applink.

Some users will probably choose the convenience of not adapting to a new concept such as this framework over the privacy benefits it provides. We assume this to be especially true for users who are not privacy-conscious.

7.6 Generalizability

This section outlines the experience of implementing different kinds of applications using Applink. Applink works well when used in conjunction with CRDTs in applications that can tolerate eventual consistency, such as the notes application.

7.6.1 Notes Application

The notes application is a good fit for Applink. Notes can be represented by the Yjs text primitive and sending links to other users to share notes is an interaction pattern that is

[20]`https://support.discord.com/hc/articles/360039213771` (last visited 2024-08-28)

[21]`https://help.steampowered.com/en/faqs/view/7EFD-3CAE-64D3-1C31#qrlogin` (last visited 2024-08-28)

also used by centralized note applications such as OneNote[22].

The notes application allows users to just start creating notes without having to sign up or log in. All notes can be synchronized to a different device or web browser by copy/pasting a link to that device to pair it once.

A screenshot of the notes application is shown in Figure 7.10. The notes application is currently deployed at `https://privdb.org`.

The requirements of the notes application have been met as the following list shows:

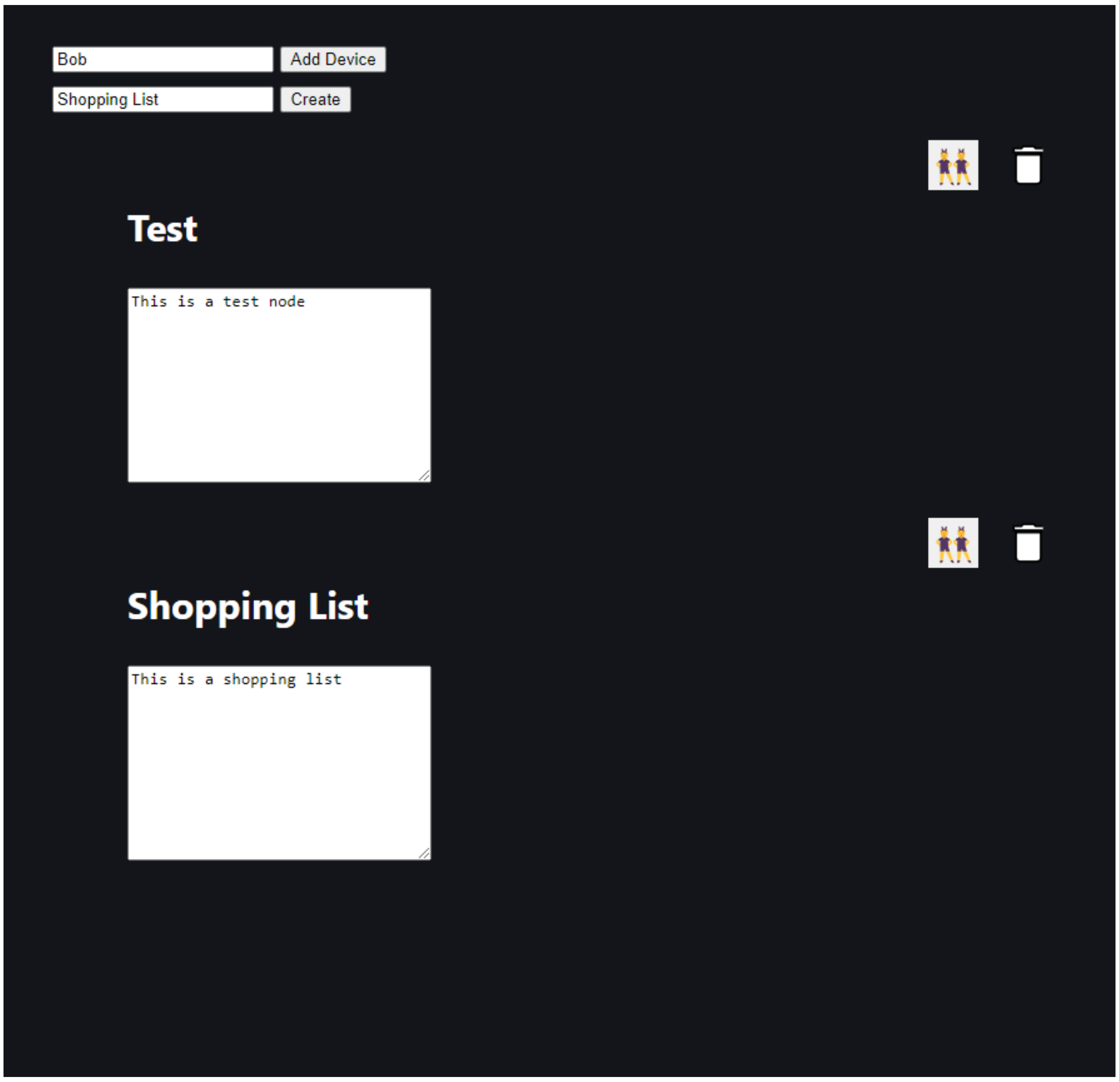
1. The application allows a user to create, update, and delete a note.
2. The application allows a user to view their notes.
3. The application allows a user to share their note with other users (0..*). Sharing a note is as easy as sharing a link in a centralized application. However, synchronization of notes is only possible when both users are online at the same time.
4. The application allows a shared note to be edited by all users it has been shared with. This is achieved by using Yjs as the underlying data structure for the notes.
5. The application allows a user to synchronize their notes with other devices. This is achieved by sharing the note with the other device and both devices (or a transitory device) being online at the same time.
6. The application does not allow users to view, modify, or delete the notes of other users without explicit permission. This can be achieved by validating the userIDs of incoming requests, which this application does.

The last two items of the requirements of the notes application include two more points that have not been fully met.

(7) The application shall not reveal the set of users with access to a note to users without access to the note. A shared note can be inferred by a user who knows the addresses of both peers that share a note and is part of the DHT network path between the peers. One possible solution to this is to add onion routing to Applink. However, as discussed in section 4.4, this is out of scope for this project.

(8) The application shall not allow anyone to prevent anyone else from using the application. It is difficult to completely eliminate the possibility of a denial-of-service attack. However, Applink provides some censorship resistance, making it more difficult to restrict the availability of an application for individual users.

[22]`https://www.microsoft.com/de-at/microsoft-365/onenote/digital-note-taking-app` (last visited 2024-08-28)



Test

This is a test node

Shopping List

This is a shopping list

Figure 7.10: Notes application

7.6.2 Beer Credit System

The main problem in implementing the BCS is keeping track of a user's current balance. In Blockchain systems such as Bitcoin, all transactions are known to all peers. Because only administrators are allowed to know about all transactions, synchronizing the transaction list between administrators is key.

Consider the case of an administrator who comes online to confirm one credit transaction and goes offline immediately after granting this transaction. No other administrator is aware of the user's increase in balance, and all of them will refuse purchases until the granting administrator is back online and shares this data with the other administrators.

One solution to this problem is to have one administrator permanently online. However, this defeats all privacy advantages of a decentralized system such as this framework. Another possible solution would be to keep track of the user's balance and item prices in a smart contract. However, this would move all functionality into the blockchain and diminish the need for Applink.

One use case for Applink in the BCS could be a customer support chat. This would work well as it is sufficient for one administrator to be online to answer a customer's question, and consistency is not a hard requirement for this type of application.

Therefore, the BCS is not a good fit for Applink. A possible workaround is to use a blockchain to process transactions, but this diminishes the need for Applink.

The requirements of the BCS cannot be fully implemented. Here, the list of requirements and their status is listed:

1. The application allows a user to read their account balance.
2. The application allows a user to request a deposit. This is even possible when no administrator is currently online, as the request will be rebroadcast when an administrator comes online.
3. The application allows a user to buy a drink.
4. The application allows a user to retrieve and store invoices in an invoice application of choice. This is a bit clunky because the user must open the invoice application and the BCS application at the same time to synchronize the invoices.
5. The application allows administrative users to approve a deposit. This is only possible when the approving administrator is online at the same time as the user requesting the deposit.
6. The application allows administrative users to view the account balances of any user. This can only be consistent when all administrators are online at the same time.
7. The application allows administrative users to modify the set of administrators.

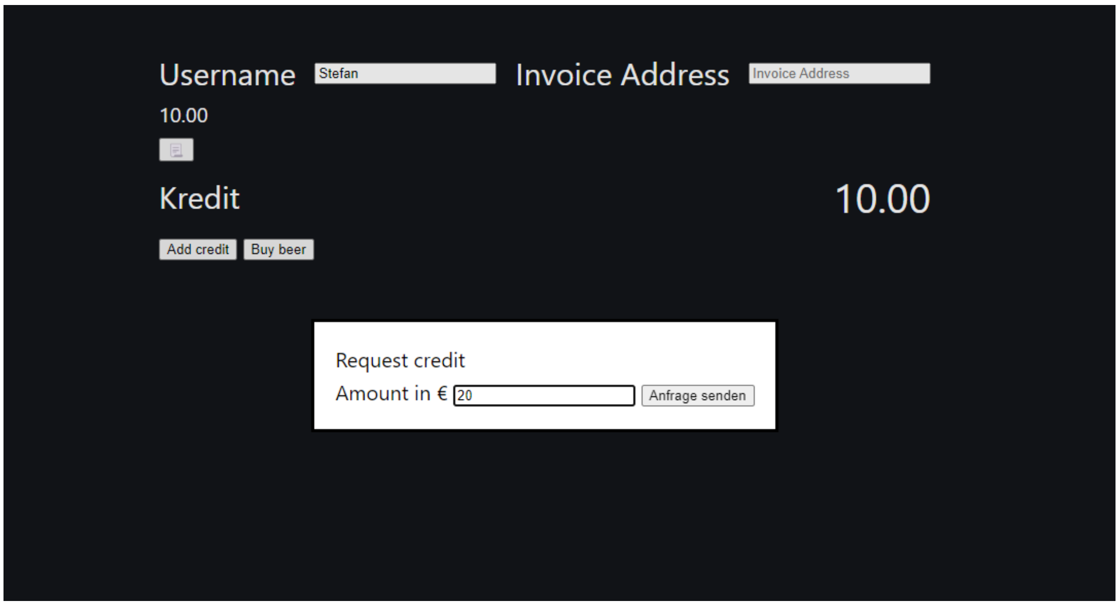


Figure 7.11: Beer Credit System – User requesting credit

8. The application allows administrative users to bulk download the already existing invoices. This is possible as long as the administrative user has synchronized all transactions with other administrators.
9. The application does not allow a user to read another user's balance. This can be achieved by validating the userIDs of incoming requests, which this application does.
10. The application shall not allow a user to buy a drink with insufficient credit. This could be violated if administrators have an inconsistent view of the user's balance.

As explained above, requirements 10 and 7 cannot be implemented within Applink.

The user flow of the BCS, where a user first requests a deposit, is shown in Figure 7.11. The admin view of the BCS of the request is shown in Figure 7.12.

7.6.3 Invoicing System

The limiting factor for the usability of the invoicing system is the constraint that an application can only receive data while it is open. Therefore, we designed the flow of receiving invoices to be a manual user request. This is quite cumbersome. The conventional way of delivering invoices by email seems much more flexible and practical.

1. The application allows a user to view a list of their invoices. This is possible with all invoices being even available offline.

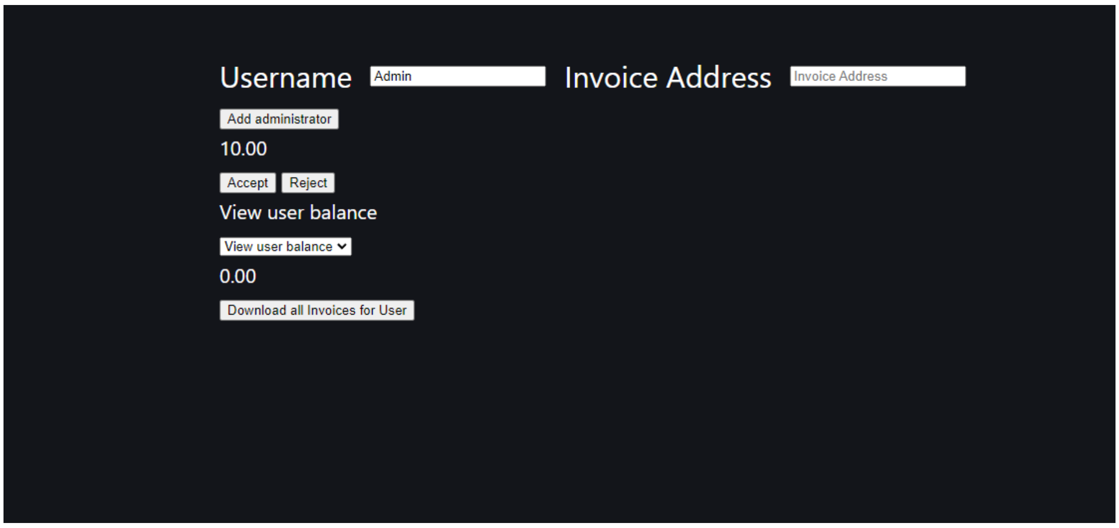


Figure 7.12: Beer Credit System – Admin View

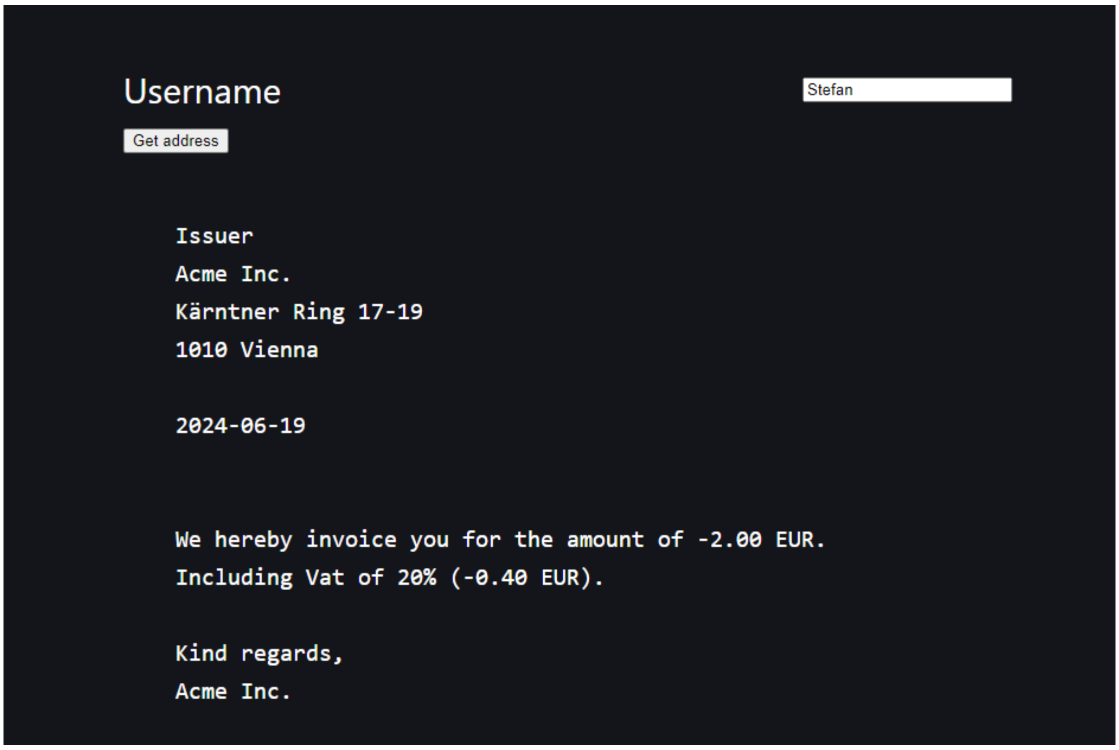
2. The application allows a user to view an invoice. This is possible with all invoices being even available offline.
3. The application allows an external application to store an invoice on behalf of the user. This is possible as long as the invoice system is open at the same time the external application attempts to store it.
4. The application prevents a user from reading invoices issued to other users. This can be achieved by validating the userIDs of incoming requests, which this application does.
5. The application offers anonymity for the sender and recipient of invoices for other users of the system and external observers. This is the case with the caveats described in subsection 7.2.4.

A screenshot of the invoicing system is shown in Figure 7.13.

7.7 Research Questions

In this section, we evaluate the research questions stated in section 1.2: (1) Evaluating if Applink can be implemented in current web browsers without extensions: Applink cannot be fully implemented without relying on centralized proxy servers for push messages that can potentially be blocked by a censoring entity.

We attempt to propose mitigations to this limitation, such as having a list of proxy servers that forward messages to the push infrastructure and making it possible to self-host such proxy servers and share these self-hosted proxies only with trusted contacts.



```
Issuer
Acme Inc.
Kärntner Ring 17-19
1010 Vienna

2024-06-19


We hereby invoice you for the amount of -2.00 EUR.
Including Vat of 20% (-0.40 EUR).

Kind regards,
Acme Inc.
```

Figure 7.13: Invoicing System

Applink can be implemented without web browser extensions. However, the WebRTC connection establishment time can be unacceptably slow on some devices. This can be mitigated by shortening the ice-gathering stage of the WebRTC connection, as shown in Listing 7.4. This workaround works by checking if the ice-gathering has yielded any candidates after 10 seconds and returning the candidates if they are available.

(2) Explore and document methods to ensure data consistency in a partially replicated system with the constraints of a web application: The notes application demonstrates that sharing Yjs documents between users is a viable way to maintain multiple documents that are shared with a set of users. The set of users a document is shared with can be defined for every document. The DhtTransportManager and `RPCInterface` abstractions allow application developers to build robust interactions between peers to resynchronize application state when a user comes online. CRDTs and Event Sourcing are two methods to achieve robust eventual consistency for applications built with Applink.

(3) The WebAuthn API is only implemented in the Chrome web browser and therefore not yet an option. However, it would allow Applink to encrypt all application data using a security key, preventing other applications with access to the web browser's storage from stealing information or impersonating users.

(4) Evaluate the performance of Applink compared to centralized web applications: Applink performs on-par or better compared to centralized applications when it comes

to page-load times.

However, the time until an initial connection is established can be significantly longer than loading even poorly performing centralized web applications. This is due to the WebRTC connection establishment time.

Once a connection is established, Applink performs well in terms of synchronization overhead and is on-par with the user experience of centralized applications.

7.8 Limitations

We designed Applink and its APIs and implemented all sample applications and have therefore a biased view about the ease-of-use of Applink despite our best efforts to be critical. We did not implement a large application using Applink, which could lead to unforeseen complexities arising in larger applications not present in the sample applications.

Furthermore, we did not test the scalability of Applink with a large number of users. Testing was restricted to a small number of users in Vienna, Austria. However, we did validate the fundamental scaling behavior in a simulation as described in subsection 7.3.3.

Applink's censorship resistance relies on the push infrastructure, which introduces a single point of failure controlled by a central entity. The push infrastructure can theoretically be blocked by a censor, and it is not possible to determine the reactions of censors in advance.

Generally, the security and especially the censorship resistance of a framework like the one described in this work can only be tested over time with real-world usage, which we have not yet done.

Designing a good reputation system for the peer-to-peer network was not within the scope of this work. Without a reputation system in place, it is much easier to launch an eclipse attack on specific peers in the network.

In this chapter, we have evaluated the security, performance, developer usability, end-user usability, and generalizability of Applink, as well as the research questions stated in section 1.2. We have also outlined the limitations of this work. In the next chapter, we will discuss future research directions.

CHAPTER 8

Future Research

In this chapter, we discuss the new research questions that have arisen from the work presented in this thesis.

8.1 Censorship Resistance

The censorship resistance of Applink has not been tested in a real-world scenario. The mechanisms we designed to resist censorship have not yet been tested in a practical censorship scenario. This testing can only take place with a wide deployment of Applink where it is actively attacked by a censor. Specifically, our assumption that the push infrastructure will not be censored needs to be tested in practice.

8.2 Developer Experience

The speed of development and robustness of applications written in Applink by developers not involved with Applink's development have not been evaluated. We identify three questions for future research:

(1) How easy is it for developers to build applications with Applink? This question can be evaluated in a user-study.

(2) What are common pitfalls? This question can be answered once more developers build applications with Applink.

(3) Is it possible to sufficiently test applications before rolling them out to significantly reduce the need for monitoring and debugging? This question can be answered by a survey of developers using Applink.

8.3 Accessing the Tor Network

Applink could potentially be used to provide a transport for the Tor network. This would allow users to access Tor hidden services or the public internet through an onion circuit without the need to download the Tor Browser Bundle. This would require the Rust implementation of the Tor protocol called Arti[1] to be compilable to WebAssembly.

8.4 Applications of the Framework in Combination with Blockchain Technology

Hash Time Locked Contracts (HTLCs) [101] could be a good fit for Applink and allow for secure payment channels. Given the authentication and encryption mechanisms of Applink, HTLCs could be used for anonymous micro-payments for content such as news articles.

Blockchains can also be used to establish the time order of transactions. Consider a ticket auction application where the order of bids is important. The event information could be shared via Applink, and just the bidding process could be implemented on the blockchain. Furthermore, a commit-reveal scheme could be implemented where the reveal phase is done via Applink.

8.5 User Experience

The user experience of applications built with this framework has not been evaluated. Whether users prefer applications built with this framework over traditional web applications remains an open question.

Applink in its current form does not include any functionality to export backups of the database, neither for users nor for developers. This is a feature that should be implemented in the future. One significant drawback for users is that they have to manage backups themselves, which could be a hurdle for non-technical users.

In this chapter, we have discussed the new research questions that have arisen from this work. In the next chapter, we will discuss our conclusions.

[1]`https://gitlab.torproject.org/tpo/core/arti` (last visited 2024-08-28)

CHAPTER 9

Conclusions

The modern web landscape necessitates robust privacy and security measures, particularly as centralized infrastructures become more prone to censorship and control. This thesis explored the potential of leveraging existing web technologies – namely the WebRTC API, the Crypto.subtle API, and the Push API-to develop a decentralized, privacy-first web application framework. Despite inherent limitations and reliance on certain centralized components, our framework demonstrates the feasibility of deploying censorship-resistant peer-to-peer applications within modern web browsers.

Our research introduced an abstraction for peer-to-peer remote procedure calls (RPCs), facilitating authenticated and authorized communications between users. This mechanism supports the development of applications that maintain data confidentiality and integrity, even in the absence of a central authority.

The successful implementation and evaluation of a notes application, utilizing conflict-free replicated data types for synchronization, highlighted the practical viability of our framework. The notes application effectively showcased Applink's ability to support real-time collaboration and offline availability.

Key contributions of this thesis include: *(1) Framework Design and Implementation:* We designed and implemented a peer-to-peer web application framework that leverages modern web technologies to enhance user privacy and data security. *(2) Abstraction for Peer-to-Peer RPCs*: Our framework introduces a mechanism for secure and authenticated remote procedure calls, supporting decentralized application functionality. *(3) Sample Applications and Evaluation*: We developed and evaluated sample applications, demonstrating Applink's advantages and limitations compared to centralized web applications. *(4) Performance and Security Analysis*: Evaluations of functionality, performance, and security demonstrate Applink's strengths and identified areas for further optimization.

While our framework represents a step forward in the development of decentralized web applications, several challenges and areas for future research remain. These include real-world testing of censorship resistance mechanisms and exploring mechanisms for privacy-preserving monitoring.

List of Figures

List of Algorithms

Bibliography

[1] Council of European Union, “Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec,” 2016. `https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=celex:32016R0679` (last visited 2024-08-28).

[2] H. Barrigas, D. Barrigas, M. Barata, P. Furtado, and J. Bernardino, “Overview of facebook scalable architecture,” in *Proceedings of the International Conference on Information Systems and Design of Communication (ISDOC)*, ISDOC ’14, (New York, NY, USA), pp. 173–176, Association for Computing Machinery, 2014. ISBN: 9781450327138.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP ’07, (New York, NY, USA), pp. 205–220, Association for Computing Machinery, 2007. ISBN: 9781595935915.

[4] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual understanding of microservice architecture: Current and future directions,” *SIGAPP Appl. Comput. Rev.*, vol. 17, pp. 29–45, jan 2018.

[5] A. Acquisti, C. Taylor, and L. Wagman, “The economics of privacy,” *Journal of economic Literature*, vol. 54, no. 2, pp. 442–92, 2016.

[6] C. Deußer, S. Passmann, and T. Strufe, “Browsing unicity: On the limits of anonymizing web tracking data,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 777–790, 2020.

[7] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, “Sok: Cryptographically protected database search,” in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 172–191, 2017.

[8] K. Lewi and D. J. Wu, “Order-revealing encryption: New constructions, applications, and lower bounds,” in *Proceedings of the 2016 ACM SIGSAC Conference on*

Computer and Communications Security, CCS ’16, (New York, NY, USA), pp. 1167–1178, Association for Computing Machinery, 2016. ISBN: 9781450341394.

[9] D. X. Song, D. Wagner, and A. Perrig, “Practical techniques for searches on encrypted data,” in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 44–55, 2000.

[10] R. Poddar, S. Wang, J. Lu, and R. A. Popa, “Practical volume-based attacks on encrypted databases,” in *2020 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 354–369, 2020.

[11] T. Wilde and T. Hess, “Methodenspektrum der wirtschaftsinformatik: Überblick und portfoliobildung,” tech. rep., 2006.

[12] J. Nielsen, *Usability engineering*. Morgan Kaufmann Publishers Inc., 1994. ISBN: 978-0125184069.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[14] B. Prünster, A. Marsalek, and T. Zefferer, “Total eclipse of the heart–disrupting the InterPlanetary file system,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3735–3752, 2022.

[15] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, “Eclipse attacks on Bitcoin’speer-to-peer network,” in *24th USENIX security symposium (USENIX security 15)*, pp. 129–144, 2015.

[16] M. Tran, I. Choi, G. J. Moon, A. V. Vu, and M. S. Kang, “A stealthier partitioning attack against bitcoin peer-to-peer network,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 894–909, IEEE, 2020.

[17] A. Bienstock, P. Rösler, and Y. Tang, “Asmesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’23, (New York, NY, USA), pp. 1–15, Association for Computing Machinery, 2023. ISBN: 9798400700507.

[18] M. R. Albrecht, R. Eikenberg, and K. G. Paterson, “Breaking bridgefy, again: Adopting libsignal is not enough,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 269–286, 2022.

[19] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, “Applied federated learning: Improving google keyboard query suggestions,” 2018.

[20] T. Wink and Z. Nochta, "An approach for peer-to-peer federated learning," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 150–157, 2021.

[21] A. Luqman, A. Chattopadhyay, and K.-Y. Lam, "Membership inference vulnerabilities in peer-to-peer federated learning," SecTL '23, (New York, NY, USA), Association for Computing Machinery, 2023. ISBN: 9798400701818.

[22] I. Goldberg, D. Wagner, and E. Brewer, "Privacy-enhancing technologies for the internet," in *Proceedings IEEE COMPCON 97. Digest of Papers*, pp. 103–109, 1997.

[23] P. Syverson, R. Dingledine, and N. Mathewson, "Tor: The secondgeneration onion router," in *Usenix Security*, pp. 303–320, USENIX Association Berkeley, CA, 2004.

[24] A. Bahramali, A. Bozorgi, and A. Houmansadr, "Realistic website fingerprinting by augmenting network traces," CCS '23, (New York, NY, USA), pp. 1035–1049, Association for Computing Machinery, 2023. ISBN: 9798400700507.

[25] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 269–282, 2009.

[26] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig, "Hornet: High-speed onion routing at the network layer," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1441–1454, Association for Computing Machinery, 2015. ISBN: 9781450338325.

[27] C. Kuhn, M. Beck, and T. Strufe, "Breaking and (partially) fixing provably secure onion routing," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 168–185, 2020.

[28] F. Schillinger and C. Schindelhauer, "End-to-end encryption schemes for online social networks," in *Security, Privacy, and Anonymity in Computation, Communication, and Storage* (G. Wang, J. Feng, M. Z. A. Bhuiyan, and R. Lu, eds.), (Cham), pp. 133–146, Springer International Publishing, 2019. ISBN: 978-3-030-24907-6.

[29] F. Schillinger and C. Schindelhauer, "Concealed communication in online social networks," in *Applied Cryptography in Computer and Communications* (B. Chen and X. Huang, eds.), (Cham), pp. 117–137, Springer International Publishing, 2021.

[30] B. Dodson, I. Vo, T. Purtell, A. Cannon, and M. Lam, "Musubi: Disintermediated interactive social feeds for mobile devices," in *Proceedings of the 21st International Conference on World Wide Web (WWW)*, WWW '12, (New York, NY, USA), pp. 211–220, Association for Computing Machinery, 2012. ISBN:9781450312295.

[31] Y. Hu, A. Trachtenberg, and P. Ishwar, “Collaborative privacy for web applications,” in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 460–469, 2019.

[32] F. Wang, R. Ko, and J. Mickens, “Riverbed: Enforcing user-defined privacy constraints in distributed web services,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 615–630, USENIX Association, Feb. 2019. ISBN: 978-1-931971-49-2.

[33] Y.-D. Bromberg, Q. Dufour, D. Frey, and E. Rivière, “Donar: Anonymous VoIP over tor,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 249–265, 2022.

[34] H. Shafagh, L. Burkhalter, S. Ratnasamy, and A. Hithnawi, “Droplet: Decentralized authorization and access control for encrypted data streams,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2469–2486, 2020.

[35] D. Kogan and H. Corrigan-Gibbs, “Private blocklist lookups with checklist,” in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 875–892, 2021.

[36] R. Khandelwal, T. Linden, H. Harkous, and K. Fawaz, “PriSEC: A privacy settings enforcement controller,” in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 465–482, 2021.

[37] R. Khandelwal, A. Nayak, H. Harkous, and K. Fawaz, “Automated cookie notice analysis and enforcement,” in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1109–1126, 2023.

[38] R. Khare and R. Taylor, “Extending the representational state transfer (rest) architectural style for decentralized systems,” in *Proceedings. 26th International Conference on Software Engineering (ICSE)*, pp. 428–437, 2004.

[39] J. T. K. Lo, E. Wohlstadter, and A. Mesbah, “Imagen: Runtime migration of browser sessions for javascript web applications,” in *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, WWW ’13, (New York, NY, USA), pp. 815–826, Association for Computing Machinery, 2013. ISBN: 9781450320351.

[40] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes, “Poking a hole in the wall: Efficient censorship-resistant internet communications by parasitizing on webrtc,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 35–48, 2020.

[41] Z. Newman, S. Servan-Schreiber, and S. Devadas, “Spectrum: High-bandwidth anonymous broadcast,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 229–248, 2022.

[42] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson, “Blocking-resistant communication through domain fronting.,” *Proc. Priv. Enhancing Technol.*, vol. 2015, no. 2, pp. 46–64, 2015.

[43] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014. (last visited 2024-08-28).

[44] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, “Zexe: Enabling decentralized private computation,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 947–964, 2020.

[45] A. L. Goodkind, B. A. Jones, and R. P. Berrens, “Cryptodamages: Monetary value estimates of the air pollution and human health impacts of cryptocurrency mining,” *Energy Research & Social Science*, vol. 59, p. 101281, 2020.

[46] M. Bez, G. Fornari, and T. Vardanega, “The scalability challenge of ethereum: An initial quantitative analysis,” in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 167–176, 2019.

[47] C. N. Samuel, S. Glock, F. Verdier, and P. Guitton-Ouhamou, “Choice of ethereum clients for private blockchain: Assessment from proof of authority perspective,” in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–5, 2021.

[48] J. Len, P. Grubbs, and T. Ristenpart, “Partitioning oracle attacks,” in *30th USENIX security symposium (USENIX Security 21)*, pp. 195–212, 2021.

[49] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, “Nonce-Disrespecting adversaries: practical forgery attacks on GCM in TLS,” in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[50] A. Shakevsky, E. Ronen, and A. Wool, “Trust dies in darkness: Shedding light on samsung’s TrustZone keymaster design,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 251–268, 2022.

[51] T. Jager, J. Schwenk, and J. Somorovsky, “Practical invalid curve attacks on tls-ecdh,” in *Computer Security – ESORICS 2015* (G. Pernul, P. Y A Ryan, and E. Weippl, eds.), (Cham), pp. 407–425, Springer International Publishing, 2015. ISBN: 978-3-319-24174-6.

[52] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 8656, RFC Editor, aug 2008.

[53] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, “TPM-FAIL:TPM meets timing and lattice attacks,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2057–2073, 2020.

[54] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&Done: A Single-DecryptionEM-Based attack on OpenSSL'sConstant-Time blinded RSA," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 585–602, 2018.

[55] D. Boneh *et al.*, "Twenty years of attacks on the rsa cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.

[56] K. Mus, Y. Doröz, M. C. Tol, K. Rahman, and B. Sunar, "Jolt: Recovering tls signing keys via rowhammer faults," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1719–1736, 2023.

[57] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 361–372, 2014.

[58] A. Takahashi and M. Tibouchi, "Degenerate fault attacks on elliptic curve parameters in openssl," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 371–386, 2019.

[59] I. Martiny, G. Kaptchuk, A. Aviv, D. Roche, and E. Wustrow, "Improving signal's sealed sender," *NDSS. The Internet Society*, 2021.

[60] G. Cherubin, R. Jansen, and C. Troncoso, "Online website fingerprinting: Evaluating website fingerprinting attacks on tor in the real world," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 753–770, 2022.

[61] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann, "World-wide web: the information universe," *Internet Research*, vol. 2, no. 1, pp. 52–58, 1992.

[62] A. Rauschmayer, *Speaking JavaScript: an in-depth guide for programmers.* O'Reilly Media, Inc., 2014. ISBN: 9781449364991.

[63] M. Squarcina, P. Adão, L. Veronese, and M. Maffei, "Cookie crumbles: breaking and fixing web session integrity," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 5539–5556, 2023.

[64] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "The security lottery: Measuring Client-Side web security inconsistencies," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2047–2064, 2022.

[65] Y. M. Kim and B. Lee, "Extending a hand to attackers: browser privilege escalation attacks via extensions," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 7055–7071, 2023.

[66] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of WebAssembly,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 217–234, 2020.

[67] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems,” in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pp. 67–72, IEEE, 1997.

[68] P. Wagle, C. Cowan, *et al.*, “Stackguard: Simple stack smash protection for gcc,” in *Proceedings of the GCC Developers Summit*, vol. 1, Citeseer, 2003.

[69] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld, “Hardening the security analysis of browser extensions,” SAC ’22, (New York, NY, USA), pp. 1694–1703, Association for Computing Machinery, 2022. ISBN: 9781450387132.

[70] S. Calzavara, S. Roth, A. Rabitti, M. Backes, and B. Stock, “A tale of two headers: A formal analysis of inconsistent Click-Jacking protection on the web,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 683–697, 2020.

[71] M. Gierlings, M. Brinkmann, and J. Schwenk, “Isolated and exhausted: attacking operating systems via site isolation in the browser,” in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 7037–7054, 2023.

[72] P. Snyder, S. Karami, A. Edelstein, B. Livshits, and H. Haddadi, “Pool-Party: Exploiting browser resource pools for web tracking,” in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 7091–7105, 2023.

[73] A. Agarwal, S. O’Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, “Spook.js: Attacking chrome strict site isolation via speculative execution,” in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 699–715, 2022.

[74] P. Chinprutthiwong, R. Vardhan, G. Yang, Y. Zhang, and G. Gu, “The service worker hiding in your browser: The next web attack target?,” in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID ’21, (New York, NY, USA), pp. 312–323, Association for Computing Machinery, 2021. ISBN: 9781450390583.

[75] S. A. Mirheidari, M. Golinelli, K. Onarlioglu, E. Kirda, and B. Crispo, “Web cache deception escalates!,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 179–196, 2022.

[76] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, “Cached and confused: Web cache deception in the wild,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 665–682, 2020.

[77] M. Rhodes-Ousley, *Information security the complete reference*. McGraw Hill Professional, 2013.

[78] T. Reddy, A. Johnston, P. Matthews, and J. Rosenberg, “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),” RFC 8656, RFC Editor, feb 2020.

[79] M. Thomson, “Message encryption for web push,” RFC 8291, RFC Editor, November 2017.

[80] A. M. I. Fette, “The websocket protocol,” RFC 6455, RFC Editor, December 2011.

[81] C. Kuhn, M. Beck, S. Schiffner, E. Jorswieck, and T. Strufe, “On privacy notions in anonymous communication,” *Proceedings on Privacy Enhancing Technologies*, vol. 2, pp. 105–125, 2019.

[82] D. L. Chaum, “Untraceable electronic mail, return addresses, and digital pseudonyms,” *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[83] A. Tran, N. Hopper, and Y. Kim, “Hashing it out in public: Common failure modes of dht-based anonymity schemes,” in *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society*, WPES ’09, (New York, NY, USA), pp. 71–80, Association for Computing Machinery, 2009. ISBN: 9781605587837.

[84] M. Schuchard, A. W. Dean, V. Heorhiadi, N. Hopper, and Y. Kim, “Balancing the shadows,” in *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*, WPES ’10, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, 2010. ISBN: 9781450300964.

[85] E. Erdin, C. Zachor, and M. H. Gunes, “How to find hidden users: A survey of attacks on anonymity networks,” *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2296–2316, 2015.

[86] B. Evers, J. Hols, E. Kula, J. Schouten, M. Den Toom, R. van der Laan, and J. Pouwelse, “Thirteen years of tor attacks,” 2016. (last visited 2024-08-28).

[87] R. B. Miller, “Response time in man-computer conversational transactions,” in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS ’68 (Fall, part I), (New York, NY, USA), pp. 267–277, Association for Computing Machinery, 1968. ISBN: 9781450378994.

[88] J. Holowczak and A. Houmansadr, “Cachebrowser: Bypassing chinese censorship without proxies using cached content,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’15, (New York, NY, USA), pp. 70–83, Association for Computing Machinery, 2015. ISBN: 9781450338325.

[89] J. Naude and L. Drevin, “The adversarial threat posed by the nsa to the integrity of the internet,” in *2015 Information Security for South Africa (ISSA)*, pp. 1–7, 2015.

[90] D. M'Raihi, J. Rydell, M. Pei, and S. Machani, "TOTP: Time-Based One-Time Password Algorithm," Tech. Rep. 6238, May 2011.

[91] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pp. 386–400, Springer, 2011.

[92] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Yjs: A framework for near real-time p2p shared editing on arbitrary data types," in *Engineering the Web in the Big Data Era: 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings 15*, pp. 675–678, Springer, 2015.

[93] M. Overeem, M. Spoor, and S. Jansen, "The dark side of event sourcing: Managing data conversion," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 193–204, 2017.

[94] H. Rameder, "Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools," 2021.

[95] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 82–96.

[96] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, (New York, NY, USA), pp. 331–340, Association for Computing Machinery, 2022. ISBN: 9781450392266.

[97] R. Anderson, "Chat control or child protection?," *arXiv preprint arXiv:2210.08958*, 2022.

[98] K. Nassiri and M. Akhloufi, "Transformer models used for text-based question answering systems," *Applied Intelligence*, vol. 53, no. 9, pp. 10602–10635, 2023.

[99] Y. Liu, W. Lu, S. Cheng, D. Shi, S. Wang, Z. Cheng, and D. Yin, "Pre-trained language model for web-scale retrieval in baidu search," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, (New York, NY, USA), pp. 3365–3375, Association for Computing Machinery, 2021. ISBN: 9781450383325.

[100] G. K. Zipf, "Relative frequency as a determinant of phonetic change," *Harvard studies in classical philology*, vol. 40, pp. 1–95, 1929.

[101] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Stabilization, Safety, and Security of Distributed Systems* (A. Pelc and A. A. Schwarzmann, eds.), (Cham), pp. 3–18, Springer International Publishing, 2015. ISBN: 978-3-319-21741-3.

Appendix

Overview of Generative AI Tools Used

We use the GPT-4 and GPT-4o models from OpenAI[1] to suggest spelling and grammar corrections of this entire work. Relevant commits to the thesis repository contain the text “gpt” or “GPT”. The prompt used for the suggestions is:

```
Please correct the spelling and grammar of this document:
---
<text>
---
```

where <text> is the text to be corrected. The suggestions are reviewed and manually applied to the document.

Furthermore, we used Github Copilot[2] to suggest code snippets for the implementation of Applink and the sample applications. The code has been reviewed and adapted to fit the requirements of Applink.

[1] `https://chatgpt.com/` (last visited 2024-08-28)
[2] `https://github.com/features/copilot` (last visited 2024-08-28)